图灵电子与电气工程丛书

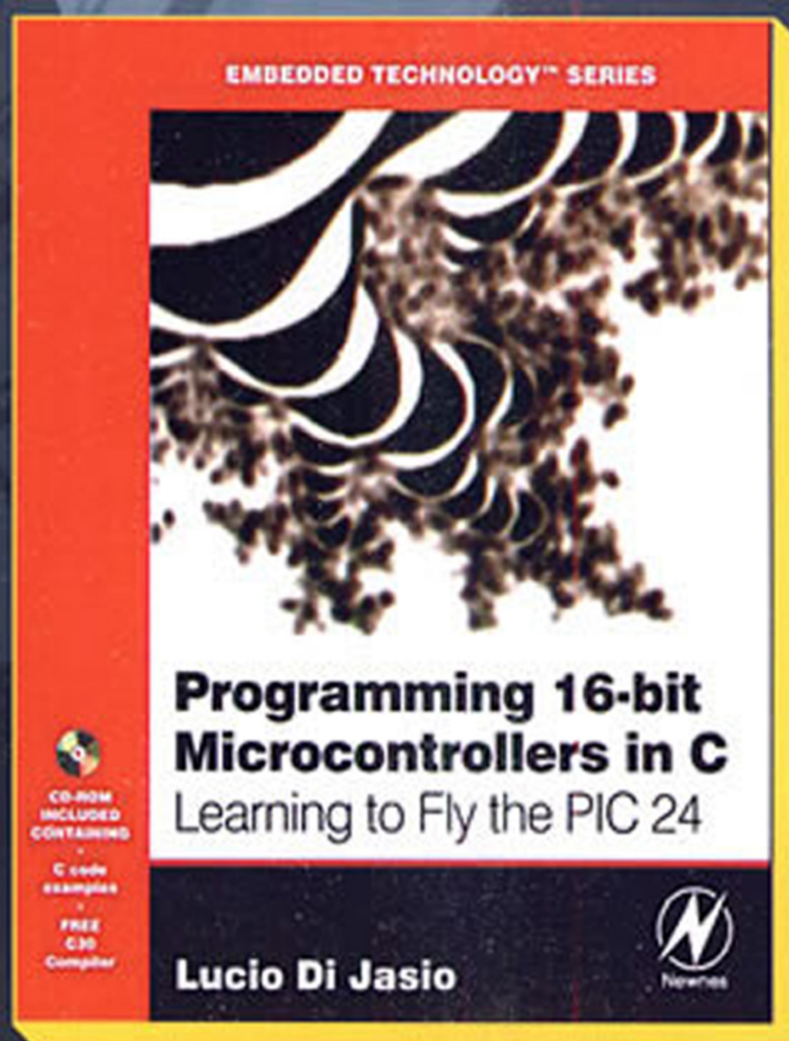
16位单片机C语言编程

基于PIC24

Programming 16-bit Microcontrollers in C

Learning to Fly the PIC 24

[意] Lucio Di Jasio 著
李中华 张雨浓 黄晓红 译



人民邮电出版社
POSTS & TELECOM PRESS



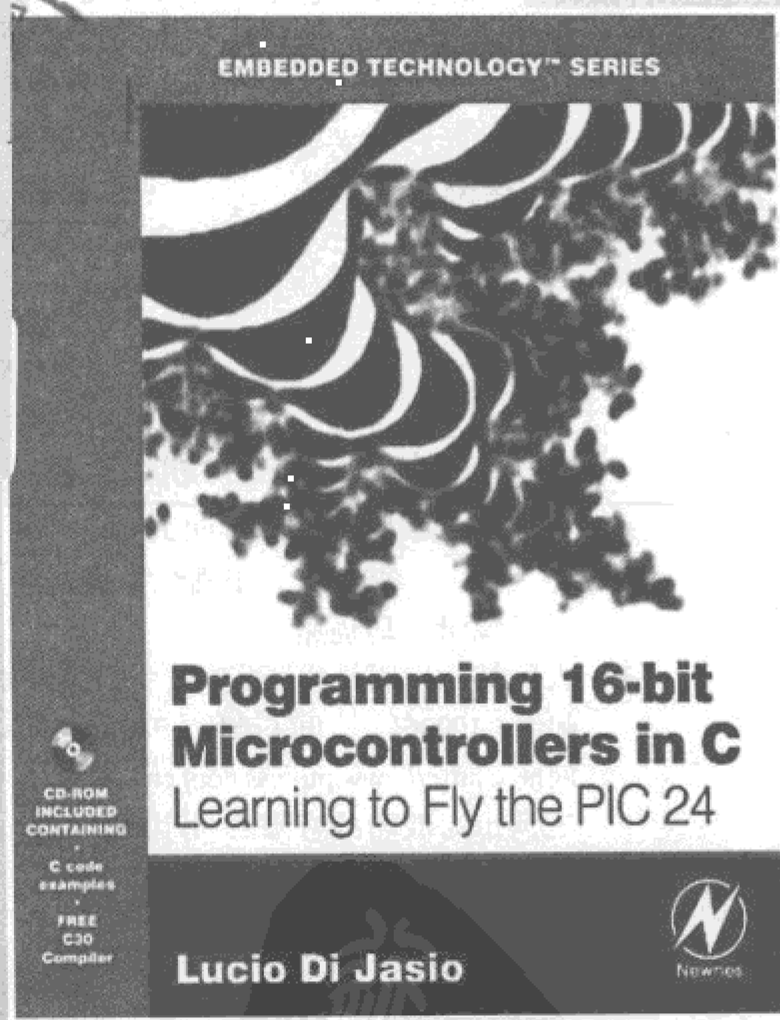
图灵电子与电气工程丛书

16位单片机C语言编程

基于PIC24

Programming 16-bit Microcontrollers in C

Learning to Fly the PIC 24



人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

16位单片机C语言编程：基于PIC 24/（意）贾西欧（Jasio, L. D.）著；李中华，张雨浓，黄晓红译. —北京：人民邮电出版社，2010.4
（图灵电子与电气工程丛书）
ISBN 978-7-115-22149-0

I. ①1… II. ①贾… ②李… ③张… ④黄… III. ①单片微型计算机－C语言－程序设计 IV. ①TP368.1 ②TP312

中国版本图书馆CIP数据核字（2010）第007537号

内 容 提 要

本书是关于16位PIC微控制器C语言编程的经典著作，采用飞行员训练教程的模式，历经从“首次飞行”至“自由翱翔”的全训练过程。全书围绕PIC 24微控制器应用系统设计的C语言描述，从PIC 24微控制器的基本C编程语法开始，涵盖了PIC 24微控制器中断处理、存储器分配、通信接口、人机接口、视频处理、外围部件接口等模块的功能原理和C程序实现等内容。

本书即可作为高等院校相关专业本科生、研究生的课程教材，也可供从事微控制器应用设计和嵌入式系统开发的工程技术人员参考。

图灵电子与电气工程丛书
16位单片机C语言编程：基于PIC 24

◆ 著　　［意］Lucio Di Jasio
　译　　李中华　张雨浓　黄晓红
　责任编辑　马晓燕
◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
　邮编　100061　电子函件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　三河市海波印务有限公司印刷
◆ 开本：787×1092　1/16
　印张：19
　字数：472千字　2010年4月第1版
　印数：1-3 000 册　2010年4月河北第1次印刷
著作权合同登记号　图字：01-2009-2898号
ISBN 978-7-115-22149-0

定价：49.00元

读者服务热线：（010）51095186　印装质量热线：（010）67129223
反盗版热线：（010）67171154

版 权 声 明

Programming 16-bit Microcontrollers in C: Learning to Fly the PIC24, by Lucio Di Jasio, ISBN: 978-0-7506-8292-3.

ISBN: 978-981-272-185-3.

Elsevier (Singapore) Pte Ltd.
3 Killiney Road
#08-01 Winsland House I
Singapore 239519
Tel：(65)6349-0200
Fax：(65)6733-1817

First Published 2010
2010 年初版



引　　言

一直以来，我几乎天天都在想着写一本书，写写我这一辈子最钟爱的一件事——驾驶飞机遨游蓝天！希望通过这本书激励其他工程师能像我一样敢于冒险、实现梦想——学习飞行，成为私人飞机驾驶员。然而，有限的实际飞行经历还不能让我成为一名值得信赖的飞行专家。因此，当有机会撰写一本关于 Microchip 最新的 16 位 PIC24 微控制器的书籍时，我忍不住想尝试将编程和飞行结合起来。毕竟，学习飞行也要遵循一个成熟的训练过程，即人们熟悉新技能并超越自身极限的一个历程。它通常引导你通过一定的理论学习和实际操作，才能获得初级飞行员的资格。飞行员资格实际上只是一个崭新冒险过程的起点，有人说那是继续学习的资格。其实，飞行学习的过程和学习新的编程技巧或者掌握新型微控制器功能的过程是极其相似的。

我将这两个学习领域的平行式比拟贯穿于全书，并在每一章的参考文献中也介绍一些飞行读物。如果读者真的有这种飞行梦想，希望本书能激发起读者的好奇心，给读者以梦想成真的学习动力。

读者定位

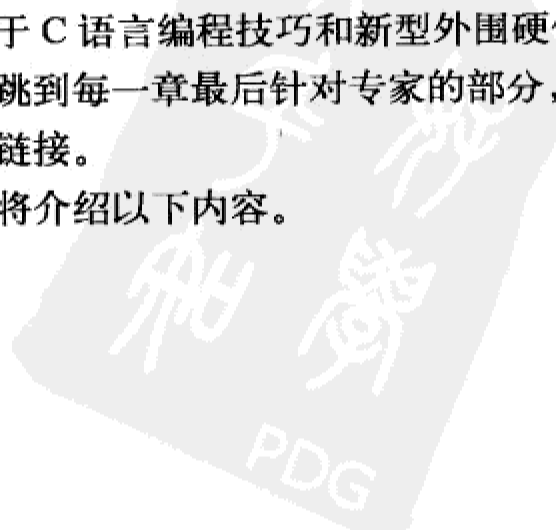
我本该在这里告诉读者：在阅读本书的时候，你将会体验到很多有趣的软件和硬件实验，而且会学习到如何在全新的 16 位 RISC 处理器上从零开始使用 C 语言编程。但是，说实话，我没法这么说，因为这不是十分准确。真心希望读者在阅读本书的时候能够体验到更多的乐趣、感受更多的趣味实验。不过，读者必须做些准备工作并努力学习，才能消化本书内容，经过前几章介绍后内容难度会很快加大。

本书是为具有初级和中级编程能力的人员编写的，不适合纯粹意义上的“新手”。因此，本书不会从最基础的二进制数、十六进制符号以及编程基础知识开始讲授。不过，在介绍难度较大的项目之前，本书将简单地介绍 C 语言的编程基础，因为它和最新的通用 16 位微控制器应用密切相关。本书尤其适合下列 4 类人员。

- 嵌入式控制程序员：具有基于汇编语言的微控制器编程经验，但对 C 语言编程只有基本的认识。
- PIC 微控制器专家：对 C 语言编程有基本的了解。
- 学生或专业人员：对 PC 的 C（或 C++）编程有一定知识。
- 其他高手：鉴于程序员不喜欢被简单地分类，所以特意为读者创造了这个类别！

不同层次和经验的读者，都可以在各章中找到感兴趣的内容。本书将尽量保证在每一章中都安排关于 C 语言编程技巧和新型外围硬件设备的介绍。如果读者对相关内容都已经熟悉了，完全可以跳到每一章最后针对专家的部分，或者思考附加练习，甚至进一步地研究/阅读参考书目和网上链接。

本书将介绍以下内容。



- ❑ 嵌入式控制的 C 程序结构：循环，循环，再循环。
- ❑ 基本的计时和 I/O 操作。
- ❑ 使用 PIC24 的中断实现 C 语言的基本嵌入式控制的多任务。
- ❑ 新的 PIC24 外围设备（以下不分顺序）。
  - ■ 输入捕获。
  - ■ 输出比较。
  - ■ 改变通知。
  - ■ 主并行端口。
  - ■ 异步串行通信。
  - ■ 同步串行通信。
  - ■ 模数转换。
- ❑ 如何控制 LCD 显示。
- ❑ 如何生成视频信号。
- ❑ 如何生成音频信号。
- ❑ 如何访问大容量媒介。
- ❑ 如何与 PC 实现大容量设备的文件共享。

本书结构

像飞行课程一样，本书由三部分组成。第一部分由 5 个难度逐渐递增的章节组成。其中每一章都会介绍 PIC24FJ128GA010 微控制器的一个基本外围硬件设备和一个 C 语言问题。而且，在每一章中至少会开发一个演示项目。开始时，这些项目需要使用 MPLAB SIM 软件仿真器，除了可能用到 Explorer16 演示板外，不需要其他真实的硬件设备。

本书的第二部分由 5 个章节组成。因为某些外围设备需要真实硬件进行测试，所以 Explorer16 演示板（或者相似的第三方设备）在这部分会变得更重要。

本书的第三部分包括 5 个内容更为丰富的章节。每一章的内容都建立在过去章节介绍的课程之上，同时因为开发项目的复杂性提高，还增加了新的外围设备内容。本书第三部分的项目需要用到 Explorer 16 演示板和基本的焊板知识（读者可能需要用到电烙铁）。如果读者想避免麻烦，或者尚没有基本的硬件焊接工具，可以从网站 http://www.flyingthepic24.com 上找到一个能满足全部演示项目需要的电路和元件扩展板。

每一章给出的所有源代码也已包含在本书附属资源内，读者可以登录图灵公司网站（http://www.turingbook.com），免费注册后下载。

不要误解了本书

本书不能代替 Microchip 公司出版的 PIC24 数据表、参考指南和程序员手册，也不能代替 MPLAB C30 编译器的用户指南及 Microchip 提供的所有程序库和相关软件工具。整本书会经常提及上述文件和工具，而且必要时将给出方框图和摘录。本书的叙述不能代替官方网站或者用

户指南上提供的信息。读者应该注意本书的表述是否和官方文档上有分歧，必须随时参考最新材料。如果确实发现有不一致的地方，请读者一定要用电子邮件告诉我。我将不胜感激，并且会在网站 http://www.flyingthepic24.com 上发布收到的所有纠错信息和实用提示。

本书也不能作为 C 语言的初级读本。虽然在前面的部分章节中对 C 语言作了一些回顾，但是读者可以在参考资料部分列出的课程和图书中找到更完善的介绍。

备忘录

无论是专业还是业余的飞行员，在每次飞行前或者飞行中，都会按照备忘录来完成每个操作。这并不是由于那些操作步骤长得无法记忆，又或者是飞行员的记忆力比其他人差。原因在于：研究表明人的记忆是会衰退的，尤其是在承受压力的时候，因此飞行员都会使用备忘录。使用备忘录也会使他们比其他专业人士犯的错误更少，飞行员可是将保证安全看得比什么都重要的。

当然，作为程序员，在使用 PIC24 进行开发编程时，即使多做或者忘记了做什么，也不会发生什么致命危险。不过，本书还是为读者准备了一些简单的常用编程和调试任务的备忘录。希望这些备忘录能在读者刚开始学习新 PIC24 工具时提供帮助——甚至在以后像作者一样同时面对不同厂家的开发环境和多个项目时，仍然能够派上用场。

目 录

第一部分 飞行入门



第二部分 单 飞

第三部分 跨 国 飞 行

第一部分

飞 行 入 门



第 1 章　首　　飞

本章内容

- ▶编译和连接
- ▶创建第一个项目
- ▶端口初始化
- ▶重测 PORTA
- ▶测试 PORTB

每一个飞行学员的首飞通常都是模糊的回忆，充满着一系列短暂而强烈的感受，这包括：

- ❑ 初次起飞的冲击，尽管是由教练操作的；
- ❑ 在听了教练"能开车的人就能开飞机"的观点后，若要保持飞机直线飞行几分钟，依然会有嘴唇发白、手心冒汗的感受；
- ❑ 急性运动性眩晕，当教练回到驾驶位置执行降落任务，进行那个容易引起晕机的"侧滑"动作时，让人觉得跑道就要穿过侧面的机窗了。

对于每一个刚迈进嵌入式编程世界的新读者来说，第 1 章也会带给你类似的感觉。

1.1　飞行计划

每一次飞行都是有目的的，最好一开始就准备一份飞行计划。

本书的第一个项目是使用 16 位的 PIC24 微控制器。对于部分读者而言，可能会使用 MPLAB IDE 集成开发环境和 MPLAB C30 语言套件来开始第一个项目。即使读者以前从来没听说过 C 语言，也可能知道著名的"Hello World"编程例子。如果连这个例子也不知道，那么下面就加以简单地介绍。

第一本 C 语言的书是由 Kernighan 和 Ritchie 在几十年前编写的。从那时起，每一本正统的 C 语言书籍都会介绍一个在计算机屏幕上显示"Hello World"字样的示例程序。就算没有上千本，那么也应该有上百本的书籍延续了这个传统，本书也不例外。不过，本书会略有一些区别。实际上，本书之所以讨论可编程微控制器，是因为要设计嵌入式控制器应用。尽管可以假定任何个人电脑或工作站都会用到显示器，但在嵌入式控制领域这个假定并不成立。因此，本书的第一个嵌入式应用将关注更基础类型的输出端口——数字 I/O 引脚。在后面更深入的章节中，将会介绍 LCD 显示器和其他连接到串行口的终端设备。当然，到时将会给出比"Hello World"更有趣的实验。

1.2　飞前备忘录

每一次飞行都会有飞前检查——绕着飞机走走，检查油缸里的汽油以及机身上的机翼等。因此，这里应首先检查所需的组件是否已经准备和安装好（可从 Microchip 网站 http://www.

microchip. com/mplab 下载最新版本）：

- MPLAB IDE，免费的集成开发环境；
- MPLAB SIM，软件仿真器；
- MPLAB C30，C 编译器（免费的学生版）。

接下来，要按照下面这个“创建新项目”备忘录，使用 MPLAB IDE 创建一个新项目。

(1) 选择“Project→Project Wizard”激活新项目向导，它会通过下面的步骤自动引导读者操作。

(2) 选择 PIC24FJ128GA010 器件，单击 Next。

(3) 选择 MPLAB C30 编译器套件，单击 Next。

(4) 创建一个新文件夹，命名为“Hello”。将项目命名为“Hello Embedded World”，单击 Next。

(5) 简单地单击下一个对话框的 Next——因为不需要从以前的项目或目录下复制任何源文件。

(6) 单击 Finish，完成向导设置。

由于是第一个项目，所以还需加上以下的步骤。

(7) 打开一个新的编辑窗口。

(8) 输入以下三行注释：

```
//
//   Hello Embedded World!
//
```

(9) 选择“File→Save As”，将文件保存为“`Hello.c`”。

(10) 选择“Project→Save”，保存项目。

1.3 飞行

现在要开始编写代码了。读者可能会不知所措，尤其是从来没有使用 C 语言编写过嵌入式控制应用程序代码的读者。第一行代码应该是：

```
#include <p24fj128ga010.h>
```

这并不是严格的 C 语句，但是这一伪指令会告诉编译器在继续运行程序之前，先读取设备说明文件的内容。设备说明“`.h`”文件仅仅是一个长长的列表，指明了所选 PIC24 模型中所有内部特殊功能寄存器（SFR）的名字和长度。如果 include 文件正确，则文件中寄存器的名字恰恰是设备数据表上正在使用的。如果还有疑问，可以打开文件查看——这是一个可以用 MPLAB 编辑器打开的简单文本文件。下面是 `p24fj128ga010.h` 文件的部分程序段，其中定义了程序计数器和一些其他的特殊功能寄存器（SFR）：

```
...
extern volatile unsigned int  PCL __attribute__((__sfr__));
extern volatile unsigned char PCH __attribute__((__sfr__));
extern volatile unsigned char TBLPAG __attribute__((__sfr__));
extern volatile unsigned char PSVPAG __attribute__((__sfr__));
```

```
extern volatile unsigned int  RCOUNT __attribute__((__sfr__));
extern volatile unsigned int  SR __attribute__((__sfr__));
...
```

返回到“Hello.c”源文件，加上几行，定义函数main()：

```
main()
{

}
```

尽管函数体是空的，不会执行任何操作，但它现在已经是一个完整的C语言程序了。在那两个花括号里面，很快将放入实现嵌入式控制应用的一些指令。

不管函数 main()出现在什么位置，无论是在最开始的几行，还是在一个几十万行的文件的最后几行，它都标志微控制器（程序计数器）在上电复位或者其他复位后程序开始运行的位置。

需要注意的是，在进入函数 main()之前，微控制器会执行连接器自动插入的一个较短的初始化代码段。这个代码段又被称作 c0 码。c0 码将实现基本的例行内务处理，包括微控制器栈的初始化以及其他事务。

现在的任务是启动一个或多个 I/O 引脚，即端口 A 的引脚 RA0~RA7。在汇编语言中，可以使用一些 mov 指令来将字面值（literal value）传送给输出端口。在 C 语言中，操作会变得更简单，即直接写入如下面例子中的“赋值语句”：

```
#include <p24fj128ga010.h>

main()
{
    PORTA = 0xff;
}
```

首先要注意的是，每一个 C 语句都是以分号结束的。另外，C 语句同数学方程很相似，不过它不是数学方程！

赋值语句的右边会先被计算。所得的结果（在这个例子中仅仅是一个字面值常量）会被保留，然后传送到左边的接收容器。在这个例子中，接收容器是微控制器的一个 16 位特殊功能寄存器（已在.h 文件中定义过）。

> **注解** 在 C 语言中，如果字面值前面有 0x，则表明它是十六进制数。否则编译器将假设其为默认的十进制数。相似地，0b 表示二进制数，而因为历史的原因，单个的“0”表示八进制数。（现在还有人使用八进制数吗？）

1.3.1 编译和连接

现在，我们已经写出了 main()函数，也是第一个 C 程序的唯一的函数。下面，怎样把源程序转换成可执行的二进制代码呢？

那就使用 MPLAB 集成开发环境（IDE）吧，它很容易上手！只要用鼠标轻轻一点就可以啦。这个操作被称作项目构建。一系列又长又复杂的工序主要由以下两大步骤组成。

- ❑ 编译：激活 C 编译器，并生成目标代码文件（.o）。这个文件暂时还不能执行。当大部分代码生成时，所有的函数和变量的地址仍然是未定义的。实际上，这又叫作可重定位的代码目标。如果有多个源文件，对每个文件都会重复地执行这个步骤。
- ❑ 连接：激活连接器，并在内存空间中为每个函数和变量分配适当的位置。同时，任意数量的预编译器目标代码文件和标准库函数都会在这个时候根据需要增加进来。连接器生成的几个输出文件实际上都是二进制可执行文件（.hex）。

以上的这些步骤，在读者单击项目（Project）菜单的选项“Build All”后，就会以很快的速度执行完成。

如果选择命令行界面，读者会欣喜地发现，除了使用 MPLAB IDE 外，还可使用其他的方法来激活编译器和连接器，从而获得同样的结果，不过，还需要查找 MPLAB C 编译器的用户指南上的指令。本书的后续章节将一直使用 MPLAB IDE 界面和适当的备忘录来简化操作。

为了让 MPLAB 知道哪个（或哪些）文件需要编译，应该把它（们）的名字（在本例中是 Hello.c）添加到项目的源文件列表（Source Files List）中。

为了使连接器能正确地为每个变量和函数分配地址，需要向 MPLAB 提供指定设备的“连接器脚本”文件（.gld）的名称。正如 include（.h）文件是用来告诉编译器指定设备的特殊功能寄存器（SFR）的名字（和大小）一样，连接器脚本（.gld）文件是用来告诉连接器内存的预定义位置（由设备数据表决定）和提供基本的内存空间信息，如闪存的可用空间大小、RAM 存储器的可用空间大小及其地址范围。

连接器脚本文件是一个简单的文本文件，可以使用 MPLAB 编辑器来打开和检查。

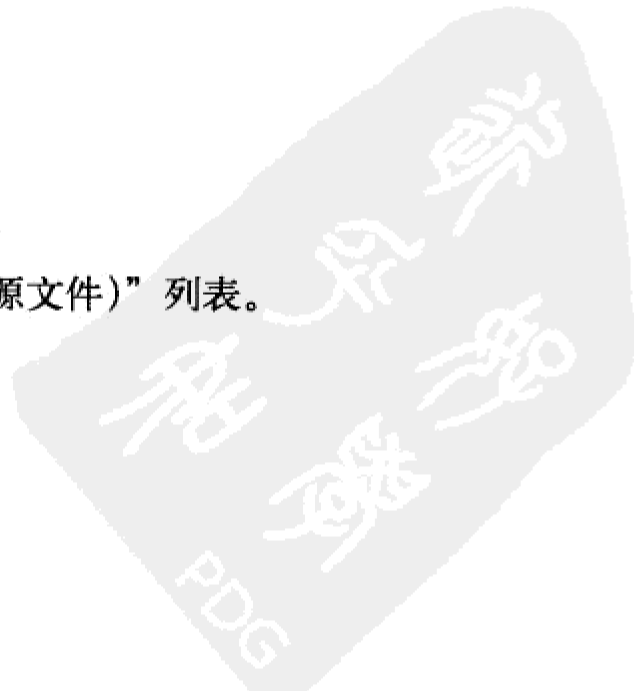
下面是 p24fj128ga010.gld 文件的程序片段，定义了程序计数器和一些特殊功能寄存器的地址：

```
 ...
 PCL          = 0x2E;
_PCL          = 0x2E;
 PCH          = 0x30;
_PCH          = 0x30;
 TBLPAG       = 0x32;
_TBLPAG       = 0x32;
 PSVPAG       = 0x34;
_PSVPAG       = 0x34;
 RCOUNT       = 0x36;
_RCOUNT       = 0x36;
 SR           = 0x42;
_SR           = 0x42;
...
```

1.3.2 构建第一个项目

首先来回顾一下完成第一个演示项目所需的几个步骤。

(1) 把当前源文件添加到“Project Source Files（项目源文件）”列表。



此时有3个备忘录可供选择，分别对应于使用3种不同的方法来实现相同的功能。在这里第一次用到的是：

(a) 打开项目窗口，如果还没打开，选择“View→Project”；

(b) 使用编辑窗口的光标，单击右键激活编辑器的弹出（pop-up）菜单；

(c) 选择“Add to project”（添加项目）。

(2) 把PIC24的“连接器脚本”文件添加到项目中。

按照下面这个“把连接器脚本添加到项目”备忘录来完成以下步骤。

(a) 右键单击项目窗口中的连接器脚本列表；

(b) 选择“Add file”（添加文件），浏览并选择MPLAB子目录 `support/gld` 下的“`p24fj128ga010.gld`”文件。

现在，项目窗口应该类似于图1-1所示。

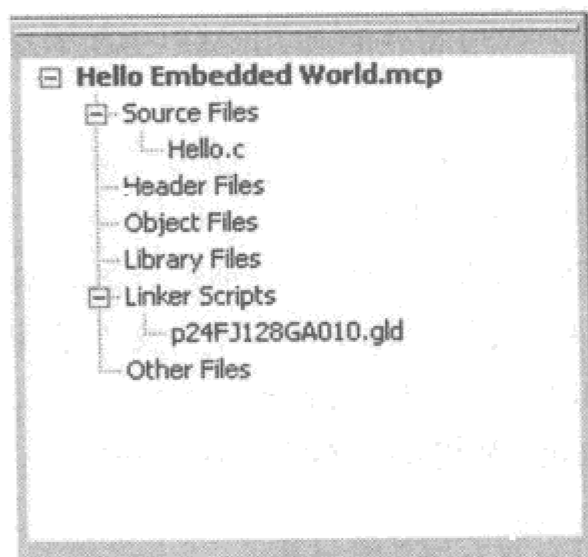


图1-1 “Hello Embedded World”项目的MPLAB IDE项目设置窗口

(3) 选择“Project→Build”功能，依次观察C30编译器和连接器的运行、生成的可执行代码以及MPLAB IDE Build窗口里的一些有用信息。

> **注解** “构建项目”备忘录包含的另外几个步骤，对于以后执行更加复杂的实例是很有帮助的。（如图1-2所示。）

(4) 选择“Debugger（调试）→Select Tool（选择工具）→MPLAB SIM（MPLAB仿真器）”，选择并激活仿真器作为本节的主要调试工具。注意：这个“MPLAB SIM调试器设置”备忘录会提示读者如何正确地配置调试器。

如果一切正常，在试运行代码之前，还要先打开Watch（监视）窗口，选择并添加 `PORTA` 特殊功能寄存器（输入或者在SFR组合框中选择 `PORTA`，然后单击“Add SFR”按钮）。（如图1-3所示。）

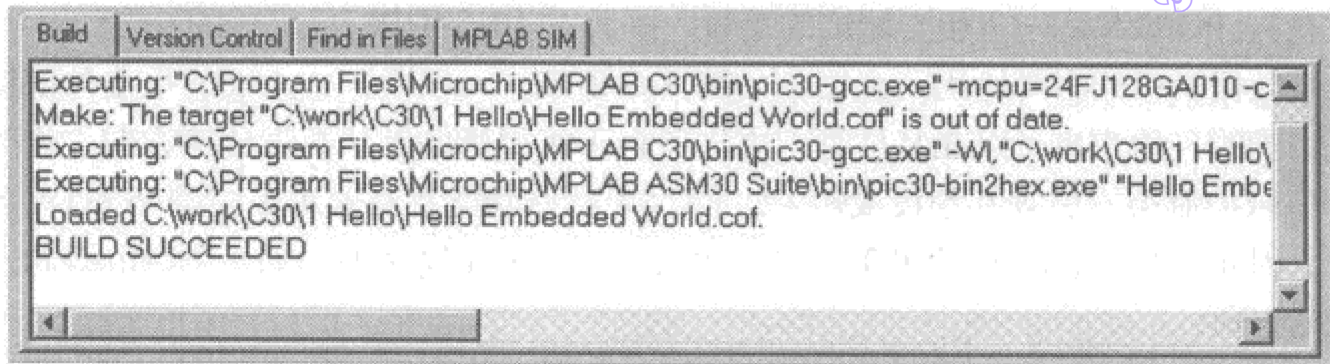


图 1-2　项目成功建立后，MPLAB IDE 输出窗口的 Build 标记信息

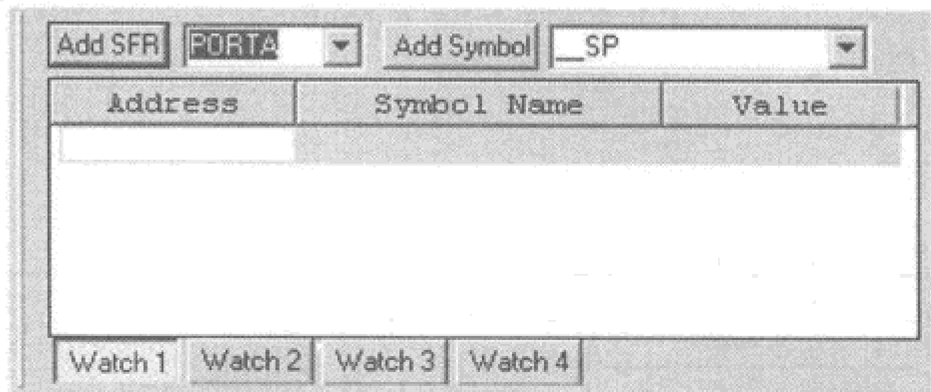


图 1-3　MPLAB IDE 的 Watch（监视）窗口

(5) 单击仿真器复位按钮（或选择“Debugger（调试器）→Reset（复位）”），观察 PORTA 的内容。复位后，PORTA 的内容应该是空的。然后，将光标置于主函数的端口分配语句行，右键打开菜单，选择“Run to Cursor”选项。（如图 1-4 所示。）

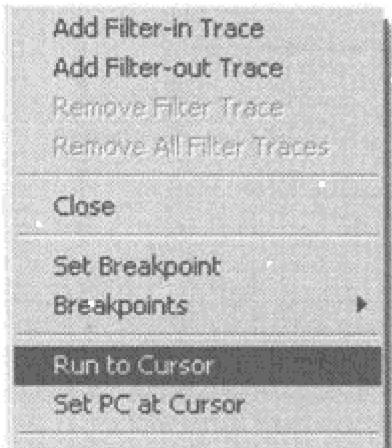


图 1-4　MPLAB IDE 编辑器菜单（右键单击）

以上操作可以跳过所有的 C 编译器初始化代码（c0），而直接进入程序代码的开始部分。

(6) 现在单步运行（使用跨过（Step-Over）或跨入（Step-In）功能）来执行本书第一个程序有且仅有的一个语句，在 Watch 窗口中观察 PORTA 的内容如何变化。或者是，观察到没有什么变化：真奇怪！

1.3.3　端口初始化

现在是言归正传的时候了，尤其要介绍 PIC24FJ128GA 的数据表（可参阅第 9 章关于 I/O

端口的信息)。PORTA 是一个相当繁忙的端口，一共有 16 个引脚。

请查看数据表中的引脚输出示意图，如图 1-5 所示。可以看到，上面的很多外部模块都是复用同一引脚的。用户也可以在复位时，决定所有 I/O 引脚的默认数据传输方向。作为标准，所有的 PIC 微控制器的引脚都预设成输入。TRISA 特殊功能寄存器用来控制 PORTA 各引脚的传输方向。因此，如果想改变 PORTA 引脚的状态，需要在程序中增加一个赋值语句，来改变它们的传输方向：

```
#include <p24fj128ga010.h>
main()
{
    TRISA = 0;          // all PORTA pins output
    PORTA = 0xff;
}
```

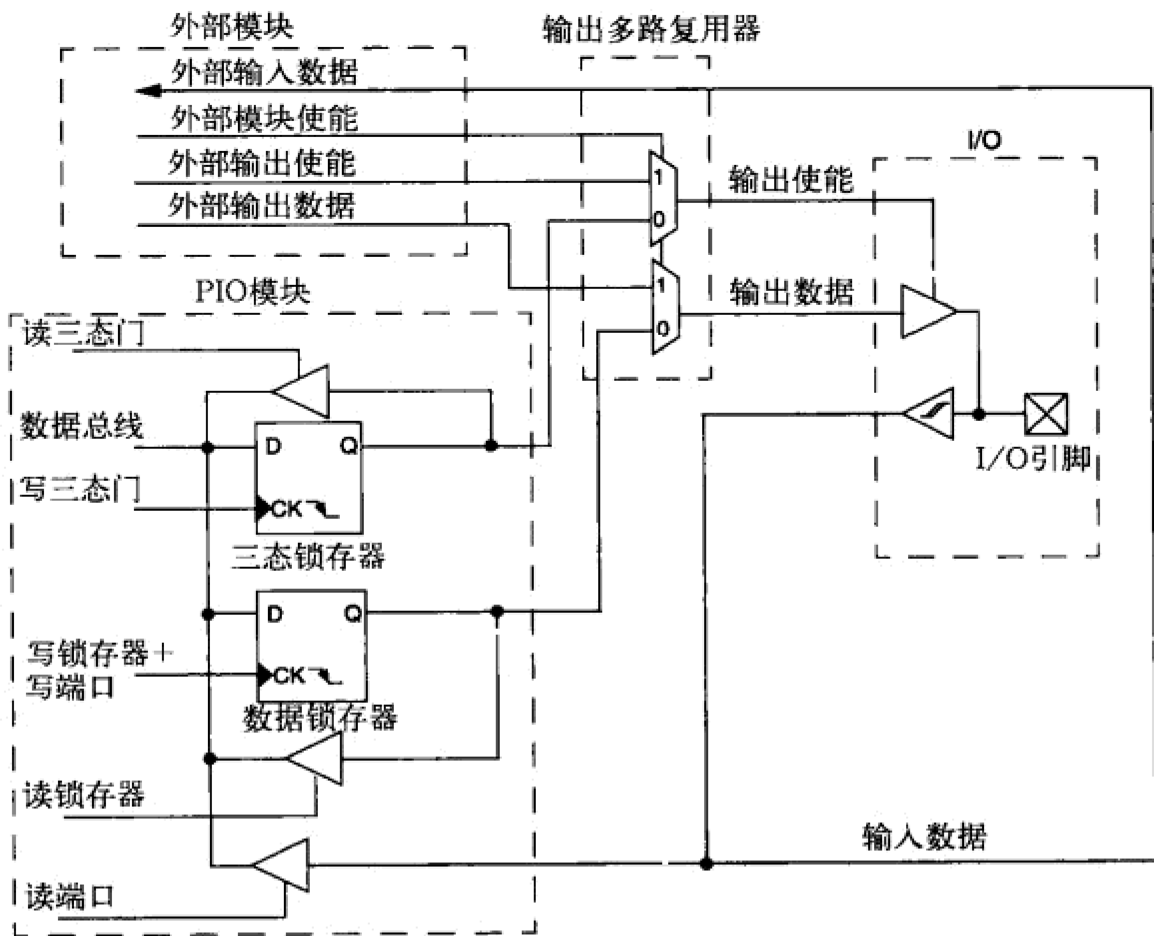


图 1-5 PIC24 典型 I/O 端口的示意图

1.3.4 重测 PORTA

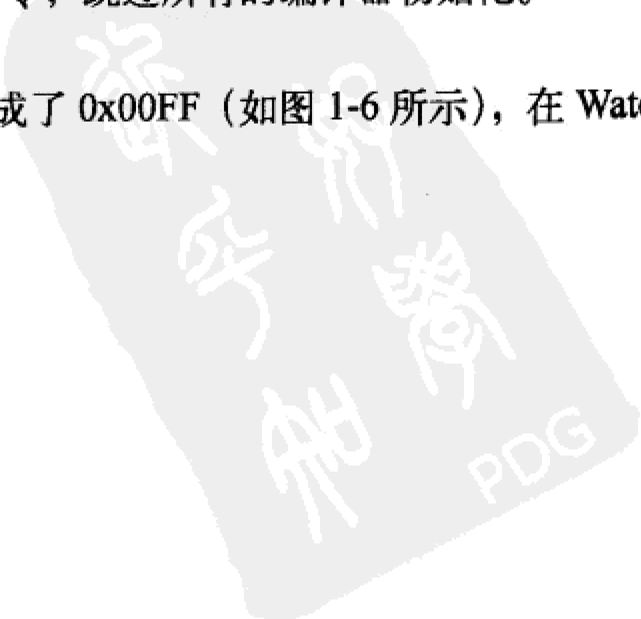
(1) 现在重新构建项目。

(2) 把光标置于 TRISA 赋值语句上。

(3) 同前面一样，执行“Run to Cursor”命令，跳过所有的编译器初始化。

(4) 执行两次单步运行……成功了！

如果一切正常，将会看到 PORTA 的内容变成了 0x00FF（如图 1-6 所示），在 Watch 窗口用红色标示出来了。Hello, World!



Address	Symbol Name	Value
02C2	PORTA	0x00FF

图 1-6 MPLAB IDE 的 Watch 窗口信息：PORTA 的内容改变了

本书首先介绍 PORTA，一部分是因为字母排序，一部分是因为基于这样一个事实，即在常用的 Explorer16 演示板中，PORTA 的引脚 RA0 到 RA7 可以很方便地连接到 8 段 LED 显示器。因此，如果读者要在一个真实的演示板上执行本例子的代码，将会满意地看到所有 LED 显示器都亮起来了，光亮而美丽！

1.3.5 测试 PORTB

在结束本次飞行训练之前，再介绍一个 I/O 端口 PORTB 的用法。

简单地编辑程序，并使用 TRISB 和 PORTB 分别代替 PORTA 的两个控制寄存器。重新构建项目，执行上一次练习的步骤……读者会有新的发现。对 PORTA 有效的代码竟然对 PORTB 不起作用！

不要慌！上面的小实验是为了让读者体验 PIC24 在代码移植上遇到的一个小小麻烦。这个经验会对读者学习和成长有帮助。

现在要返回到数据表，学习更多的 PIC24 输出引脚细节。8 位 PIC 微控制器和新型的 PIC24 微控制器在结构上有以下两点本质的区别。

- 大部分 PORTB 引脚与模数转换器（ADC）的模拟输入引脚是复用的。8 位结构的微控制器保留了 PORTA 引脚主要是为了这个目的——两种端口的角色可以互换！
- 对于 PIC24 来说，如果一个外部模块的输入/输出信号复用到一个 I/O 引脚上，只要该模块处于使能状态，那么它就可以完全控制 I/O 引脚，而不受 TRISx 控制寄存器内容的影响。对于 8 位微控制器，即使外围设备有请求，也需要用户来决定引脚的传输方向。

在默认状态下，同“模拟”输入复用的引脚，与“数字”输入端口是断开的。这正是上面例子所发生的情况。PIC24FJ128GA010 中 PORTB 的所有引脚在通电后，默认状态都是作为模拟输入的。因此，读取 PORTB 时返回的值是全 0。注意，尽管用户不能通过 PORTB 寄存器查看，然而 PORTB 的输出锁存器已被正确地设置。若确实需要校验的话，取而代之地可以查看 LATB 寄存器的内容。

为了将 PORTB 输入连接到数字输入，设置模－数转换（ADC）模块的输入是必要的。从数据表上可以看到，特殊功能寄存器 AD1PCFG 可用来控制各个引脚的模/数分配。（如图 1-7 所示。）

将特殊功能寄存器 AD1PCGF 的各位设置为 1，就能完成这个任务。全新而完整的示例程序如下：

```
#include <p24fj128ga010.h>

main()
{
    TRISB =       0;          // all PORTB pins output
    AD1PCFG =     0xffff;     // all PORTB pins digital
    PORTB =       0xff;
}
```

高字节：

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG15	PCFG14	PCFG13	PCFG12	PCFG11	PCFG10	PCFG9	PCFG8
位 15							位 8

低字节：

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG7	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0
位 7							位 0

位 15~0　PCFG15:PCFG0：模拟输入引脚配置控制位

1=对应模拟通道的引脚被设置为数字模式，I/O 端口是读使能的

0=引脚被设置为模拟模式，I/O 端口是读禁止的，A/D 采样引脚电压

图 1-7　AD1PCFG：ADC 端口配置寄存器

现在，经过编译和单步运行，就会得到预期的结果，如图 1-8 所示。

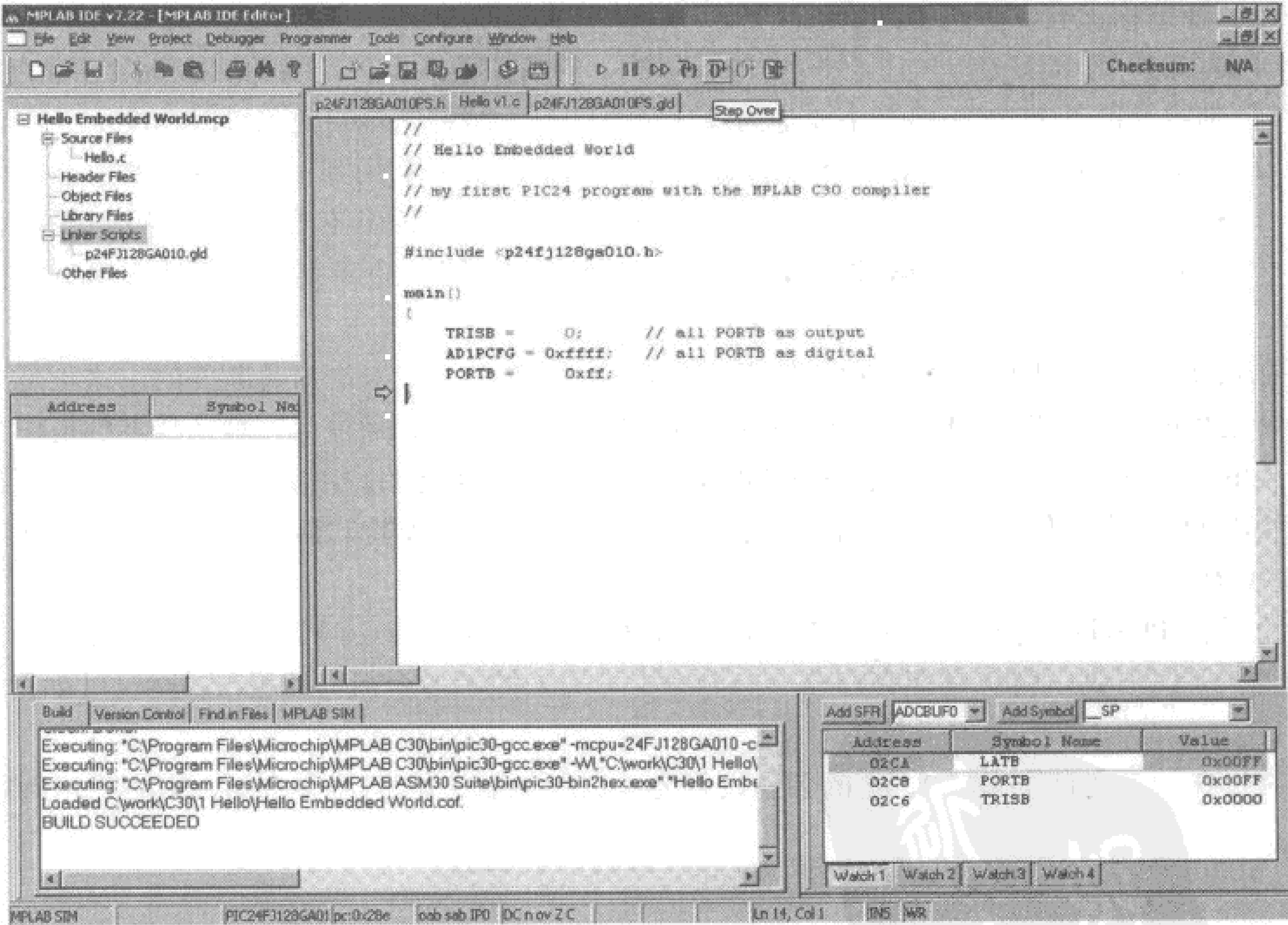


图 1-8　Hello Embedded World 项目

1.4 飞后小结

在每次飞行以后，都应该有简要的回顾。坐在舒适的椅子上，一边喝着冰水，一边和教练一起回顾在首飞中学到的东西。

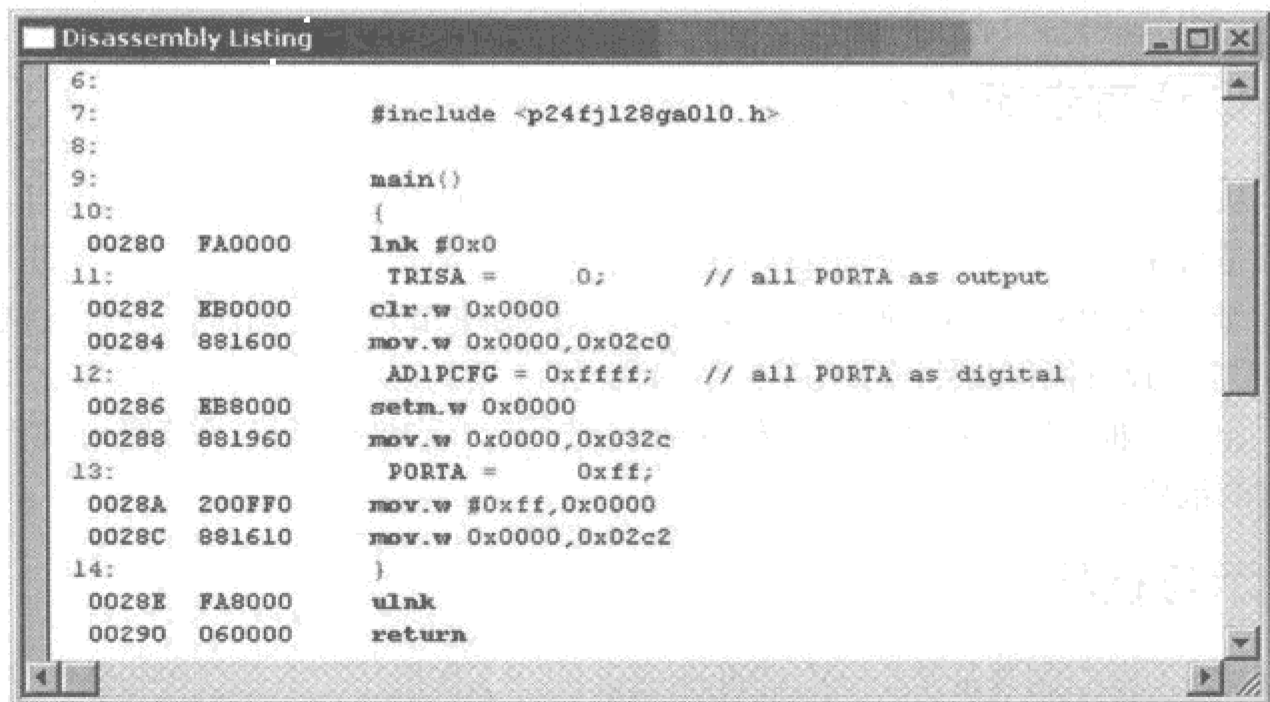
用 C 语言来编程 PIC24 微控制器是很简单的，或者说至少没有汇编语言那样复杂。根据要使用的端口，编写两条或三条指令就可以直接控制微控制器与外部世界通信的最基本工具：I/O 引脚。

当然，C30 编译器是不能读懂用户心思的。和汇编语言一样，用户要给 I/O 引脚设定传输方向。而且，用户仍需查看数据表，了解新的 16 位微控制器和已熟悉的 8 位 PIC 之间可能的区别。

尽管 C 语言已经被描述得非常神奇，但是在为嵌入式控制设备编写代码的时候，仍然需要用户对硬件的细节尽量熟悉。

1.5 给汇编语言专家的提示

如果用户觉得不能盲目地接受 MPLAB C30 编译器所产生代码的有效性，那么可能更愿意随时转到“Disassembly Listing”（反汇编列表）视图。这样，在编译器生成的每一段代码之前，都有 C 源程序注释，可以让用户快速地检查编译器生成的代码，如图 1-9 所示。




```
6:
7:                   #include <p24fj128ga010.h>
8:
9:                   main()
10:                  {
 00280  FA0000       lnk #0x0
11:                   TRISA =      0;      // all PORTA as output
 00282  EB0000       clr.w 0x0000
 00284  881600       mov.w 0x0000,0x02c0
12:                   AD1PCFG = 0xffff;    // all PORTA as digital
 00286  EB8000       setm.w 0x0000
 00288  881960       mov.w 0x0000,0x032c
13:                   PORTA =      0xff;
 0028A  200FF0       mov.w #0xff,0x0000
 0028C  881610       mov.w 0x0000,0x02c2
14:                  }
 0028E  FA8000       ulnk
 00290  060000       return
```

图 1-9 反汇编列表窗口

读者也可以通过单步运行来调试该视图里的所有代码，不过本书不建议这样做（或者说在本书的前几章中需要控制在探索性练习上）。在满足读者好奇心的同时，也要逐步学会信任编译器。最终，使用 C 语言是可以大幅度地提升编程速度、增加代码的可读性和可维护性的。

作为最后一个练习，建议读者打开内存使用检查窗口——选择“View→Memory Usage Gauge”选项，如图 1-10 所示。

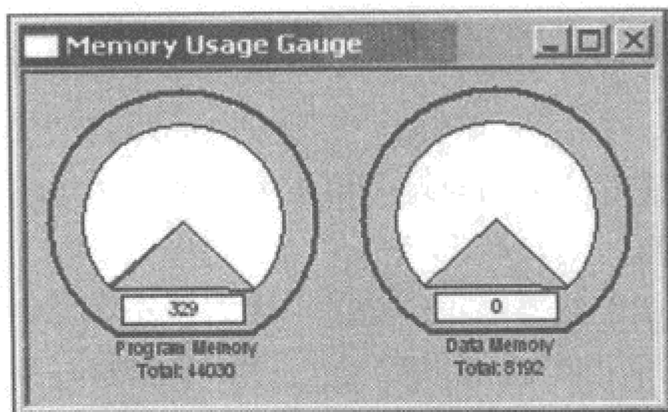


图 1-10　MPLAB IDE 的内存使用检查窗口

不要慌！尽管第一个例子只有 3 行代码，但是程序对内存的使用居然已经达到 300 多个字节，这并不能说明 C 语言的低效。（为方便起见）C30 编译器总是会产生一个最小的程序块。这就是之前曾简单提及的初始化代码（c0）。在后面讨论变量初始化、内存分配以及中断的时候，将会详细地介绍它。

1.6　给 PIC 微控制器专家的提示

对于那些已经熟悉 PIC16 和 PIC18 结构的用户来说，将会发现 PIC24 包括 I/O 端口在内的大部分控制寄存器都是 16 位宽度的。再查看 PIC24 的数据表可知，大部分外围设备的名字同 8 位外围设备的名字即使不是相同的，也是非常相似的。所以，用户立刻就会产生强烈的亲切感。

1.7　给 C 语言专家的提示

诚然，从标准 C 程序库里调用 printf 函数是可以的。事实上，程序库在 MPLAB C30 编译器里已经是可用的。不过，本书是瞄准嵌入式控制应用的，而不是为千兆字节的工作站编写代码。一定要熟悉 PIC24 微控制器中底层硬件外设的使用。一个简单的库函数调用，例如 printf，可能导致可执行文件增加几千字节的代码。不要幻想串行口和终端或者文本显示器总是可用的。相反，根据嵌入式设计领域有限的可用资源，读者应该对每个函数和每个库代码大小保持高度的敏感。

1.8　提示与技巧

PIC24FJ 系列微控制器是基于 3V CMOS 工艺的，工作电压范围是 2.0~3.6V。因此，必须使用 3V 的电源（Vdd），在输出逻辑“高”电平时，这就限制了每个 I/O 引脚的输出电压。然而，要连接到 5V 设备及应用程序是很简单的。

- 为了驱动一个 5V 的输出信号，要使用 ODCx 控制寄存器（ODCA 用于驱动 PORTA，ODCB 用于驱动 PORTB，以此类推）将每个输出引脚设置为开漏极模式，并连接外部上拉电阻到 5V 的电源。
- 数字输入引脚本身可承受 5V 电压，可直接连接至 5V 输入信号。

不过，要注意那些和模拟输入复用的 I/O 引脚——因为它们是不能承受大于 Vdd 的电压的。

1.9 练习

如果读者备有 Explorer16 实验板，那么可以进行以下操作。

(1) 使用 ICD2 调试列表来协助准备项目调试。

(2) 测试 PORTA 实例。连接 Explorer16 实验板，观察 LED0-7 的可视输出。

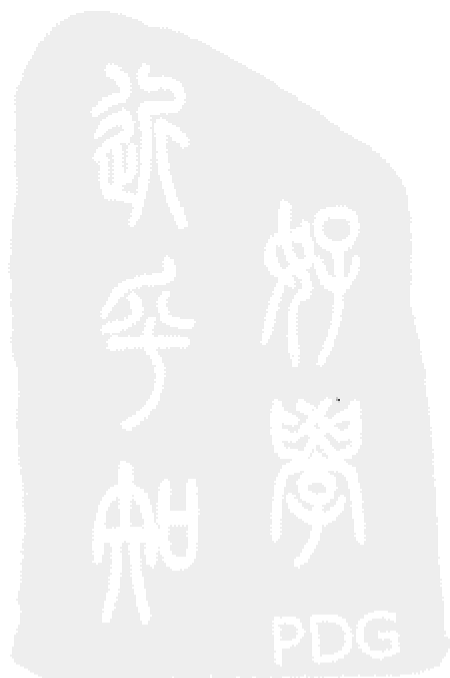
(3) 测试 PORTB 实例。将电压表（或 DMM）连接至 RB0 引脚，观察每次单步运行时指针的变化。

1.10 推荐书目

- Kernighan, B. and Ritchie, D.
  The C Programming Language
  Prentice-Hall, Englewood Cliff, NJ.
  如果看到或者听到有人在谈论“K&R”，他们就是在说这本书了！从这本被称为“白皮书”的教材在 1978 年首次出版开始，C 语言一直在不断改进中。第二版（1988 年出版）包含了更多的 ANSI C 语言标准定义，这和 MPLAB C30 编译器所支持的标准（ANSI90）很接近。
- **Private Pilot Manual**
  Jeppesen Sanderson, Inc., Englewood, CO.
  这是给每个飞行学员的参考书。即使读者对飞行仅仅是好奇，我也强烈推荐读一读。

1.11 网上链接

- http://en.wikibooks.org/wiki/C_Programming
  这是关于 C 语言的 Wiki 书籍。如果读者不介意在网上阅读的话，它的确很方便。提示：可以在标题是“A taste of C”的一章中找到无所不在的“Hello World!”练习程序。



第 2 章 模式循环

本章内容

- while 循环
- 动画仿真
- 使用逻辑分析器

所谓“模式”，就是让飞行员遵照飞行的一个标准化的矩形回路。每个机场对各条跑道的高度和位置都有规定的模式，这样可以更好地组织机场内的运作，以保证周围的航空交通不会出现混乱。假定所有的飞机都按照当时季风的特定方向飞行。飞机都具有相同的飞行高度，以便于飞行员更容易跟踪其他飞机的位置。飞机使用相同频率的无线电波，与控制塔楼（如果有的话）或者其他飞机进行通信。作为飞行学员，应该多用心，尤其在最初的课程上，跟着教练，在模式中重复练习一系列着陆即起（一触即离）的操作，让新学的技巧更为熟练。而作为嵌入式编程的学员，读者也要学习自己的循环——主循环。

2.1 飞行计划

嵌入式控制程序需要一个类似于飞行模式的结构，以便于管理代码流。本章将复习 C 语言中的基本循环语句，并适时地介绍一个新的外部模块：16 位定时器 Timer1。还会涉及 MPLAB SIM 的两个新功能：“动画”模式和“逻辑分析器”。

2.2 飞前备忘录

本章依然还是会用到之前已经安装和使用过的基本软件（可以从 Microchip 网站下载），包括：

- MPLAB IDE，集成开发环境；
- MPLAB SIM，软件仿真器；
- MPLAB C30 编译器（学生版）。

这里将再次使用 “New Project Set-up（新项目建立）”列表，使用 MPLAB IDE 创建新项目。

(1) 选择“Project→Project Wizard”激活新项目向导，开始创建一个新项目。

(2) 选择 PIC24FJ128GA010 设备，然后单击 Next。

(3) 选择 MPLAB C30 编译器套件，然后单击 Next。

(4) 创建一个新文件夹，命名为“Loop”。将项目命名为“A Loop in the Pattern”，然后单击 Next。

(5) 这里不需要从过去的项目中复制任何源文件。再次单击 Next（下一步）。

(6) 单击 Finish，完成向导设置。

接下来是“Adding Linker Script file”（添加连接器脚本文件）列表，将连接器脚本文件“p24fj128ga010.gld”添加到项目中。该文件可以从 MPLAB IDE 的安装目录“C:/Program Files/Microchip/”下的子目录“MPLAB C30/support/gld/”中找到。

继续完成“Create New File and Add to Project”（新建文件并添加到项目）列表。

(7) 新打开一个编辑窗口。

(8) 输入主程序标题：

```
//
//    A loop in the pattern
//
```

(9) 选择“Project→AddNewFileProject”，将文件保存为“loop.c”，文件就会被自动地添加到项目源文件列表中。

(10) 保存项目。

2.3 飞行

经过前一章的项目学习后，读者可能会想到这么一个问题：“当 main() 函数中的代码都执行完毕后，接着会发生什么情况呢？”其实并没有发生什么，或者说至少没有发生读者所期待的。只是设备复位，整个程序继续执行，再执行……

要注意的是，实际上，编译器将一个特殊的软件复位指令放在了 main() 函数代码的最后。对于嵌入式控制，人们总是希望从开关电源打开到关闭的时间内，应用程序都一直运行。于是，复位再执行，看似一个方便快捷的程序安排，就能让控制保持下去，除非是遇到其他意外情况。这种方法可以应用在几种有限的场合，但是读者很快就会发现，这种“循环”会带有“颠簸”。因为每次运行到程序的末尾，执行复位操作，微控制器会返回最开始执行最前面的初始化代码，包括前面简单介绍过的 c0 码段等。因此，尽管初始化部分的程序很简短，但是仍然会使得循环很不平衡。每次总是把所有的特殊功能寄存器和全局变量检查一遍，显然是不需要的，这样做也必定会降低应用的运行速度。一个更好的方法就是为程序设计一个应用程序“主循环”。首先，来复习一下 C 语言中最基本的循环代码。

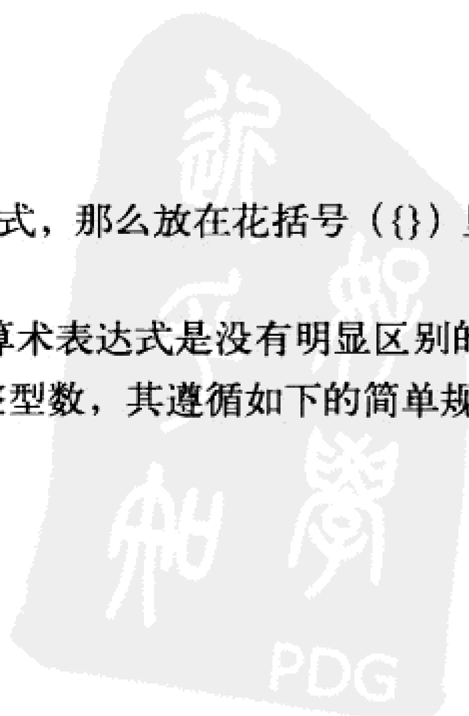
2.3.1 while 循环

在 C 语言里至少有 3 种的循环代码。以下是第一个循环——while 循环：

```
while ( x)
{
    // your code here...
}
```

只要满足括号 (x) 里的逻辑表达式，那么放在花括号（{}）里的代码就会不断地重复执行。那什么是 C 语言的逻辑表达式呢？

首先，C 语言的逻辑表达式和算术表达式是没有明显区别的。在 C 语言里，布尔逻辑真（TRUE）与假（FALSE）被表示为整型数，其遵循如下的简单规则：



- 假，用整数0表示；
- 真，可用任何非0的整数表示。

因此，1 表示为真，13 和-278 也表示为真。为了计算逻辑表达式的值，定义了一些逻辑操作符，例如以下几种。

- || 表示逻辑或操作符。
- && 表示逻辑与操作符。
- ! 表示逻辑非操作符。

根据上面的规则，这些操作符将操作数看作逻辑（布尔）量，并且返回逻辑值。下面是一些简单的例子。

（当 a=17 且 b=1 时，或者换而言之，当它们都为真时。）

- (a || b) 为真。
- (a && b) 为真。
- (!a) 为假。

还有一些操作符是用来比较数量（任何的整型数或者浮点数）大小的，然后返回逻辑值。这包括以下几种。

- == “等于”操作符。注意，它是由两个等号组成，同前面介绍的“赋值”操作符是有区别的。
- != “不等于”操作符。
- > “大于”操作符。
- >= “大于或等于”操作符。
- < “小于”操作符。
- <= “小于或等于”操作符。

下面给出一些例子。

假设 a=10，那么：

- (a>1) 为真；
- (-a>=0) 为假；
- (a==17) 为假；
- (a!=3) 为真。

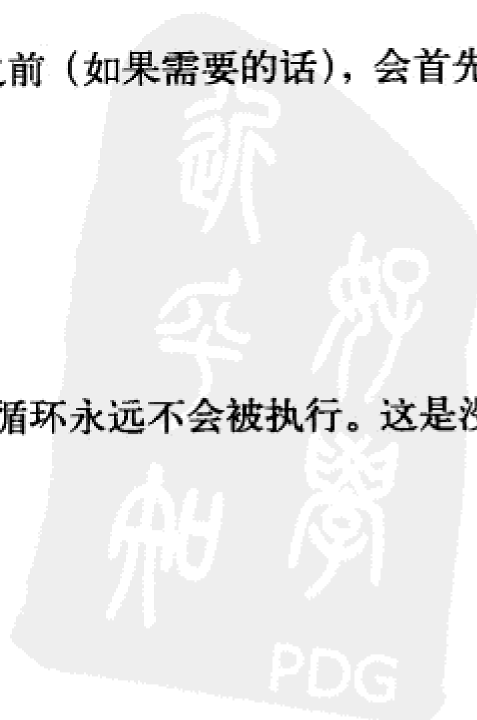
现在回到 while 循环上，只要括号里的表达式是逻辑真值（也就是非0整数），程序就会一直执行循环操作。当表达式出现了逻辑假值，循环就会终止，程序在花括号外面的第一条语句处继续执行。

注意，在执行花括号中的内容之前（如果需要的话），会首先计算逻辑表达式的值，并且在每次循环前都会重新计算。

下面是一些特殊的循环例子：

```
While ( 0)
{
    // your code here...
}
```

一个永恒的“假”，意味着这个循环永远不会被执行。这是没有意义的。实际上，它可以当



选“世界上最没用的代码”!

下面又是另一个例子:

```
while ( 1)
{
    // your code here...
}
```

一个永恒的“真”,意味着这个循环会永远执行下去。这是有意义的。而实际上,从现在开始,就要把它用在主程序的循环里。为了增加程序的可读性,有经验的工程师将规范地定义一对常量:

```
#define TRUE      1
#define FALSE     0
```

然后在程序中一致地使用它们,例如:

```
While ( TRUE)
{
    // your code here...
}
```

现在,向“loop.c”源文件里添加一些代码,并把 while 循环付诸实践。

```
#include <p24fj128ga010.h>
main()
{
    // init the control registers
    TRISA =   0xff00;// PORTA pin 0..7 as output

    // application main loop
    while( 1)
    {
        PORTA = 0xff; // turn pin 0-7 on
        PORTA = 0;           // turn all pin off
    }
}
```

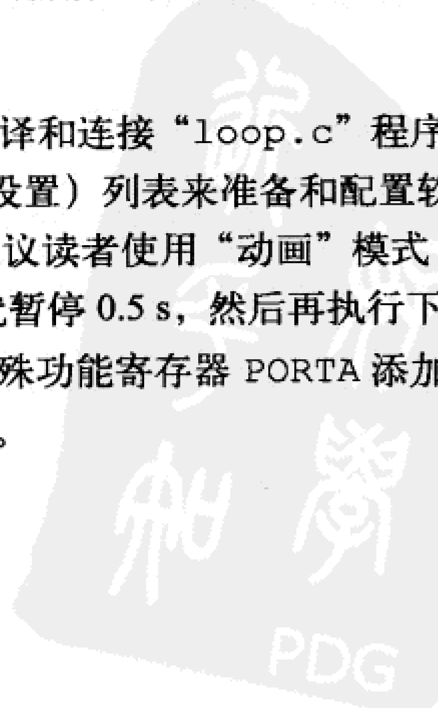
这个例子程序的结构实质上就是所有用 C 语言编写的嵌入式控制程序的结构。它们由以下两部分组成。

- ❑ 初始化,包括外围设备和变量的初始化,只在开始时执行一次。
- ❑ 主循环,包括定义应用行为的所有控制功能,将连续地执行。

2.3.2 动画模拟

使用“Project Build”(项目构建)列表编译和连接“loop.c”程序。并且使用“MPLAB SIM simulator set-up”(MPLAB SIM 仿真器设置)列表来准备和配置软件仿真器。

为了使用仿真器来测试本例中的代码,建议读者使用“动画”模式(Debugger→Animate)。在该模式下,仿真器每次执行一行 C 程序,就暂停 0.5 s,然后再执行下一行语句,这就给了读者时间来观察即时的运行结果。如果读者把特殊功能寄存器 PORTA 添加到 Watch 窗口,将会发现它的值很有节奏地在 0xff 和 0x00 之间变化。



动画模式下的执行速度可通过“Debug→Setting”（调试→设置）对话框来控制。选择“Animation/Real Time Updates”（动画/实时更新）标签，修改“Animation Step Time”（动画步进时间）参数，它的默认值是 500 ms。可以想象，动画模式是一个有用又有趣的调试工具，不过它可能会让人对真实的程序执行时间产生误解。实际上，如果将例子程序运行在真实的硬件对象上，例如 Explorer16 演示版（PIC24 的运行频率是 32 MHz），那么同 PORTA 相连接的 LED 显示器闪烁得非常快以至于人眼根本观察不到，因为每个 LED 显示器每秒钟开关了几百万次。

为了把 LED 的闪烁速度降低到每秒几次，假设使用定时器，在这个过程中可以学到如何使用集成在 PIC24 微控制器中的关键外围部件。在本例中，选取 PIC24FJ128GA010 的 5 个定时器中的第一个定时器 Timer1。它是最灵活简单的外围模块之一。读者要做的，只是快速查看一下 PIC24 的数据表，了解一些模块图和 Timer1 控制寄存器的技术细节，找出理想的初始值，如图 2-1 和图 2-2 所示。

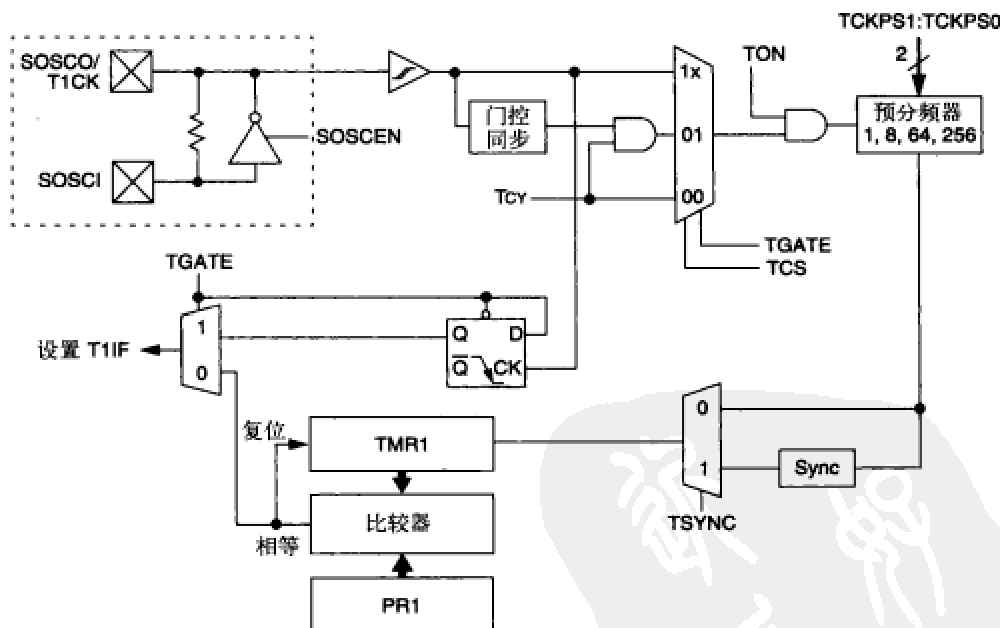
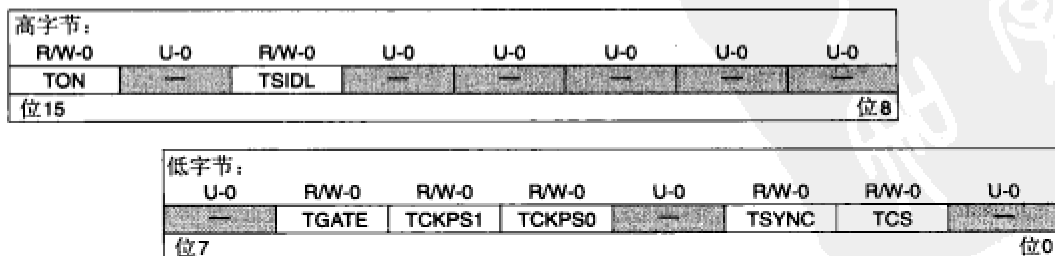


图 2-1　16 位 Timer1 方框图



高字节：

R/W-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
TON	—	TSIDL	—	—	—	—	—
位15							位8

低字节：

U-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	U-0
—	TGATE	TCKPS1	TCKPS0	—	TSYNC	TCS	—
位7							位0

图 2-2　T1CON：Timer1 控制寄存器

下面来快速学习一下控制 Timer1 大部分功能的 3 个特殊功能寄存器。

- TMR1，包括 16 位计数器值。
- T1CON，控制定时器的激活和操作模式。
- PR1，可用于产生定时器的周期性复位信号（在本例中不需要）。

将 TMR1 寄存器清零，从 0 开始计数。

```
TMR1  =  0;
```

然后，对 T1CON 寄存器初始化，让定时器具备如下的简单功能配置。

- 激活 Timer1：TON=1。
- 主 MCU 时钟作为时钟源（Fosc/2）：TCS=0。
- 预分频器设为最大值（1:256）：TCKPS=11。
- 门控输入和同步功能不要求，因为直接使用 MCU 内部时钟作为定时器时钟：TGATE=0，TSYNC=0。
- 在 IDLE 模式下没有任何动作：TSIDL=0（默认值）。

当向 T1CON 分配一个 16 位值时，就可以得到：

```
T1CON = 0b1000000000110000;
```

或者使用十六进制形式：

```
T1CON = 0x8030;
```

当定时器初始化完毕时，就进入循环，等待 TMR1 达到预设的常数值 DELAY。

```
while( TMR1 < DELAY)
{
    // wait
}
```

假设使用 32 MHz 的时钟，需要将 DELAY 设置成一个较大的数，才能让延迟时间达到 0.25 s。下面的公式可以用来计算 TMR1 循环产生的总延迟时间：

```
Tdelay = (2/Fosc) * 256 * DELAY
```

若置 Tdelay = 256 ms，解方程，得到 DELAY 的值是 16 000：

```
#define DELAY 16000
```

把两个延时循环程序放到主循环中每个 PORTA 任务前，就得到最新最好的代码例子：

```
#include <p24fj128ga010.h>

#define DELAY      16000

main()
{
    // init the control registers
    TRISA = 0xff00;               // PORTA pin 0..7 as output
    T1CON = 0x8030;       // TMR1 on, prescaler 1:256 Tclk/2

    // main application loop
```

```
    while( 1)
    {
            // 1. turn pin 0-7 on and wait for ¼ second
            PORTA = 0xff;
            TMR1 = 0;      // restart the count
            while ( TMR1 < DELAY)
            {
                    // just wait
            }

            // 2. turn all pin off and wait for ¼ second
            PORTA = 0x00;
            TMR1 = 0;      // restart the count
            while ( TMR1 < DELAY)
            {
                    // just wait
            }

    } // main loop
} // main
```

> **注解** 在使用C语言编程时，花括号的数量会随着代码的增长而快速地增加。过不了多久，即使是严格遵循缩进格式，程序员可能也无法记住闭合花括号是对应于哪个开始花括号的。在闭合花括号后，本书加上了小小的提示（注释），希望程序的易读性更好。

现在，完成整个项目的构建并验证其可行性。如果读者有 Explorer16 演示板，就可以马上运行代码。LED 显示器会以一个较舒缓的频率闪烁，大概是每秒 2 次。

如果读者想在 MPLAB SIM 仿真器上运行相同的代码，就会发现 LED 显示器闪烁得太慢了。不管读者的个人电脑速度有多快，MPLAB SIM 的执行速度都是没办法与真实的 32 MHz 的 PIC24 微控制器相比拟的。

如果读者想使用动画模式，那样会变得更糟糕。就像前面提到的，动画模式在执行每条独立代码之间都插入了 0.5 s 的延时。因此，如果只是为了调试，可以在仿真器上把 DELAY 常数的值设置得小一点（例如设为 16）。

2.3.3 使用逻辑分析器

在这次飞行结束之前，为了让实验更有趣，在构建项目后，建议读者试用一下新的仿真工具：MPLAB 逻辑分析器。

逻辑分析器提供了一个特别有效的图解式的视图，能清晰地记录设备输出引脚的值。不过在对它进行初始设置时，需要特别小心。

首先，要确定仿真器的追踪功能是打开的。

(1) 选择“Debug→Settings”（设置）对话框，然后选择 Osc/Trace 标签。

(2) 在 Tracing（追踪）选项部分，选中 Trace All（追踪全部）框。

(3) 现在从“View（视图）→ Simulator（仿真器）”逻辑分析器菜单中打开分析器窗口

(analyzer window)，如图 2-3 所示。

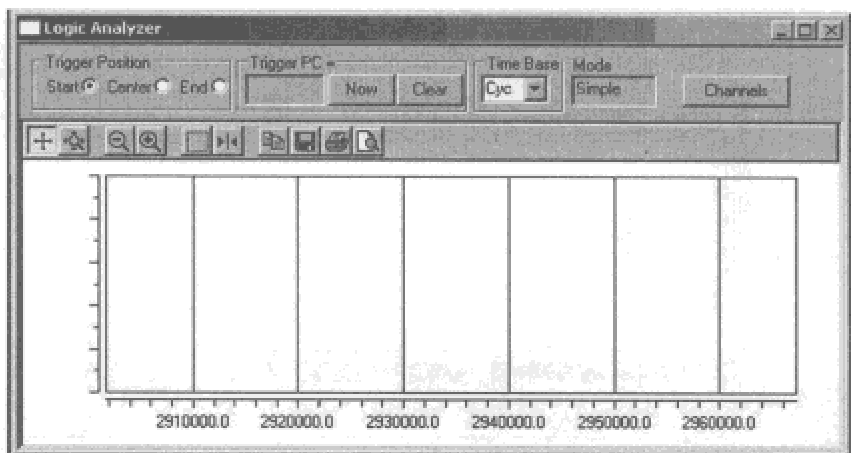


图 2-3 逻辑分析器窗口

(4) 单击通道（channel）按钮，打开通道选择（channel-selection）对话框，如图 2-4 所示。

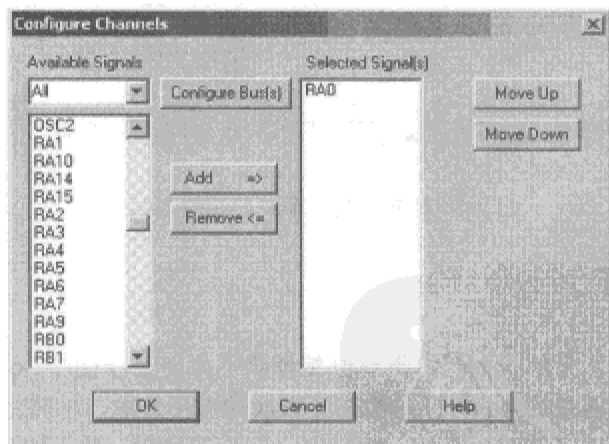


图 2-4 通道选择对话框

(5) 在这里，就可以选择要观察的输出引脚。
在本例子中，选择 RA0，然后单击 “Add=>”。
(6) 单击 “OK”，关闭通道选择（Channel-Selection）对话框。

> 注解 为了方便查阅，以上步骤全部写进了 “Logic Analyzer Set-up”（逻辑分析器设置）列表中。

单击▷按钮，运行代码一小段时间，然后单击 Halt（暂停）按钮⏸。逻辑分析器窗口就会

显示一个整齐的方波图，如图 2-5 所示。

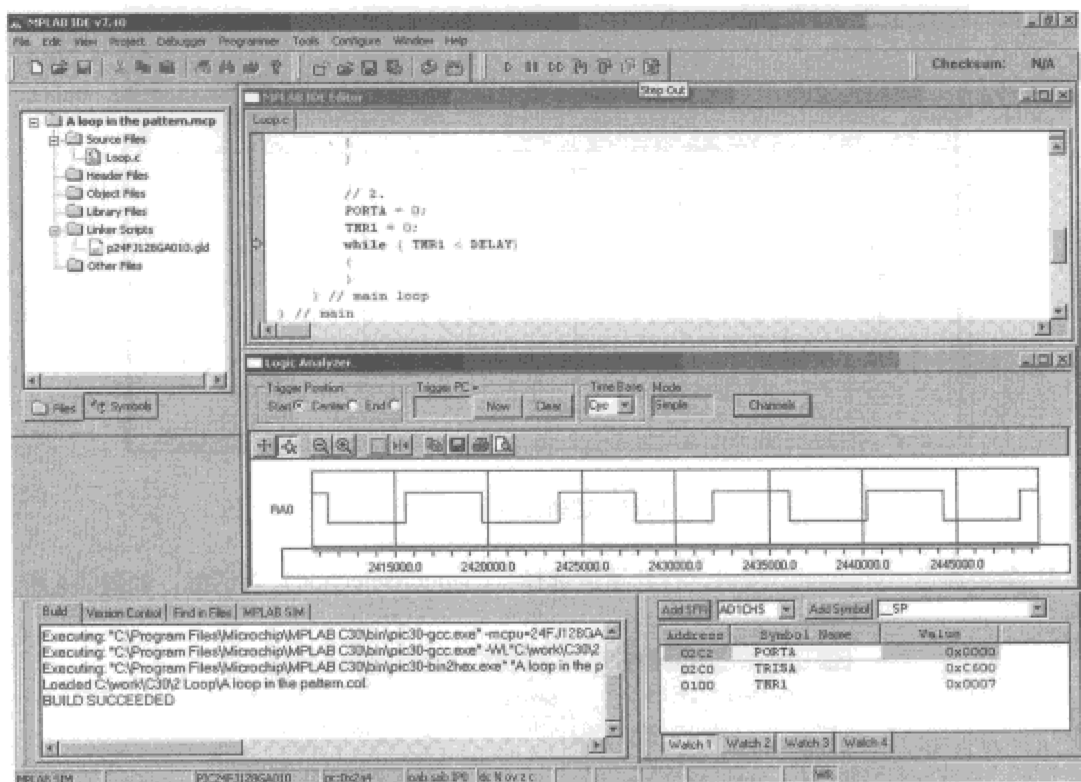


图 2-5　逻辑分析器窗口显示的方波图

2.4　飞后小结

在这简洁的一章里，介绍了 MPLAB C30 编译器是如何处理程序的。第一次给出了本章小项目的结构——初始化部分的 main() 函数与无限次的循环分离。在那之前，对 while 循环语句和逻辑表达式的求值作了简要的介绍。本章以一个例子结束，第一次使用了定时器模块，并且在逻辑分析器窗口上绘制了 RA0 引脚的输出曲线。

因为后面的章节还会涉及以上内容，所以在刚开始学习时读者不必为满腹的疑问担忧——这也是学习的一种体验。

2.5　给汇编语言专家的提示

C 语言中的逻辑表达式，对于习惯于处理有明确名字的二进制操作符（AND、OR、NOT 等）的汇编语言程序员来说，可能有些别扭。C 语言也有一套二进制操作符，不过为了避免混乱，在本章中没有提及它们。根据真值表，二进制逻辑操作符计算操作数对应数位的逻辑值。换而言之，逻辑操作符将每个操作数（忽略使用的位数）看成是单独的布尔量。

请看下面 8 位操作数的例子：

```
            11110101                        11110101 (TRUE)
二进制  OR  00001000              逻辑  OR  00001000 (TRUE)
            --------                        --------
结果        11111101              结果      00000001 (TRUE)
```

2.6 给 PIC 微控制器专家的提示

也许读者已经注意到：Timer0 不见了！好消息就是：没有人会怀念它的！实际上，PIC24 保留的 5 个定时器已经包含了所有的功能，因此没有使用 Timer0 的必要。所有控制定时器的特殊功能寄存器的命名和以前 PIC16、PIC18 微控制器相似，而且结构也非常类似。不过，还是要注意一下数据表，因为微控制器设计者加入了一些新的特性。

- ❑ 现在所有的定时器都是 16 位宽度。
- ❑ 每个定时器都有一个 16 位的周期寄存器。
- ❑ 全新的 32 位模式定时器配对机制，可用于定时器 2/3 和定时器 4/5。
- ❑ 为 Timer1 增添新的外部时钟门控特性。

2.7 给 C 语言专家的提示

如果读者习惯在个人电脑或者工作站上使用 C 编程，正如读者希望的，在 `main()` 函数结束后，控制权将交回到操作系统。尽管 PIC24 对于某些实时操作系统（RTOS）是可用的，然而这些应用都是不必要的，而且没有人会用它。这适用于本书所有的简单例子。在默认情况下，C30 编译器并不需要向任何操作系统交回控制权，而且做了最保险的事情——复位。

2.8 提示与技巧

在不被关闭或者收到复位命令的前提下，一些嵌入式应用往往被设计成主循环要经年累月地连续执行。然而微控制器的控制寄存器只是简单的 RAM 记忆单元。可能一个（掉电复位电路未检测到的）的电源波动、一个邻近噪声设备发射的电磁脉冲甚至宇宙辐射，都可能改变它们的内容，尽管这种可能性很小，但还是会有的。只要时间够长，用户就有可能在设备上发现这种情况。当要设计一个在相当长时间内运行的应用时，用户从一开始就需要认真地考虑周期性地“更新”主要外围部件的控制寄存器。

将初始化程序分成一个或者多个函数。在通电后，进入主循环前，调用这些函数。在主循环中，要保证在没有其他紧急任务挂起的时候调用初始化函数，并且每个控制寄存器都需要周期性的更新。

2.9 练习

(1) 在 PORTA 引脚输出相反值，代替开关模式。

(2) 使用翻转模式代替开关模式。

2.10 推荐书目

- Ullman, L. and Liyanage, M. (2005)
  C Programming
  Peachpit Press, Berkeley, CA
  这是一部可快速阅读的现代读本，一步步地简要介绍了 C 编程语言。
- Adams, N. (2003)
  The Flyers, in Search of Wilbur and Orville Wright
  Three Rivers Press, New York, NY
  这本书可以带你返回到历史上首次开展的动力推进飞行，在小鹰市上空仅达 120 ft(1 ft = 0.3048 m)的高度。①

2.11 网上链接

- http://en.wikipedia .org/wiki/control_flow#Loops
  给出了编程语言的概览，以及有关代码和循环编写的问题。
- http://en.wikipedia.org/wiki/Spaghetti_code
  如果不按照模式来写，代码将不受控制。

① 1903 年 12 月 17 日，莱特兄弟发明的世界第一架载人动力飞机“飞行者一号”，在美国北卡罗来纳州的小鹰市飞上蓝天。——编者注

第 3 章　更多模式，更多循环

本章内容

- do 循环
- 变量声明
- for 循环
- 更多循环示例
- 数组
- 一个新演示程序
- 使用逻辑分析器测试
- 使用 Explorer16 演示板

在航空学里，所谓“翻筋斗”（loop）是指经过高级训练的飞行员驾驶为表演任务而特殊装备的飞机执行的一种特技表演。对此，读者也许会感到信心受挫，自叹弗如，也许会感到心安理得，但有一点是可以确定的，当你还在备考初级飞行员的时候，绝不会有人要你掌握这种高难动作。尽管如此，初级飞行员还是要面临不少的挑战，比如驾驶飞机完成一系列的转弯，包括定点转弯、S 形转弯、急转弯和标准转弯。在所有这些练习里，飞行员会发现它们的难度所在——在一个三维的环境里飞行，每次只改变其中一维的参数。当围绕地面某参考点转圈时，飞行员将不得不竭尽所能地去保持恒定的高度和速度。哪怕是一丝微风都可能给保持定点距离带来困难，影响漂亮平稳的绕圈循环飞行。工多艺自熟！

在 C 语言里，也有不少的循环。什么时候使用哪个循环，怎么使用循环，也需要大量的练习来巩固，这样才能帮助读者成为更好的嵌入式控制程序员。

3.1　飞行计划

在前面的章节中，已经介绍过嵌入式控制应用程序的核心循环。本章将继续探索 C 语言中几个不同的循环方法，并且会简要地回顾整型变量声明、自增自减操作、数组声明和使用对象。同优秀的飞行课程一样，在理论之后紧接着给出实践。在本章的末尾，安排了一个有趣的练习，帮助读者更好地掌握本章学习的概念和工具。

3.2　飞前备忘录

本章将继续使用 MPLAB SIM 软件仿真器，在最后的练习中还会再次用到 Explorer16 演示板。在准备新的演示项目时，读者可以使用“New Project Set-up”（新项目建立）列表来生成新的项目，命名为“More Loops”，并且加入新的源文件“More.c”。

3.3　飞行

在 while 循环中，只要逻辑表达式返回布尔量的真值（非 0），花括号中的代码就会执行。

逻辑表达式的计算是在循环之前进行的，也就是说，如果一开始表达式返回了“非”值，那么循环体内的代码就不会被执行。

3.3.1 do 循环

如果用户需要一种循环，要求至少执行一次，后续的循环是否执行则由逻辑表达式决定，那么就应该使用另一种类型的循环了。

下面介绍 do 循环的语法：

```
do {
    // your code here...

} while ( x);
```

不要被 do 循环最后的关键字 while 所迷惑——两种循环的执行是完全不同的。

在 do 循环里，总是先执行花括号里的代码（如果有的话），然后才计算逻辑表达式的值。当然，如果希望无限次地执行 main() 函数，那么使用 do 和 while 都是一样的。

```
main()
{
    // initialization code
    ...

    // main application loop
    do {
            ...
    } while ( 1)
} // main
```

请看下面这个奇怪的例子，也许可以分析出这个循环的执行：

```
do{
    // your code segment here...
} while ( 0);
```

可以发现，上面循环里的代码执行了一次，无论内容是什么，都只执行一次。换而言之，在这里代码外的循环语句都是浪费用户打字时间的。又是“世界上最没用的代码”比赛的一位有力选手！

现在来看一个更实用的例子，使用 while 循环按预先指定的次数执行程序代码。首先，需要一个计数的变量。也就是说，会用到一个或多个 RAM 地址来存放计数值。

> 注解　在前面的两章中，几乎跳过了所有变量对象的声明，只使用了预定义变量——PIC24 的特殊功能寄存器。

3.3.2 变量声明

可以使用下面的语法定义整型变量：

```
int c;
```

由于使用关键字 int 定义了 c 为 16 位（有符号）整数，因此 MPLAB C30 编译器将给它分配两个字节的内存空间。然后，由连接器决定这两字节的内存在选定的 PIC24 模型的物理 RAM 中的位置。按照定义，变量 c 可以是从最小 –32 768 到最大 +32 767 中的值。如果需要更大的整数范围，那么可以使用 long（有符号）整数类型，如：

```
long c;
```

MPLAB C30 编译器将给变量分配 32 位（4 字节）的空间。

如果需要小一点的计数器，可以是–128 到+127 的范围，那么使用 char 整数类型就可以。

```
char c;
```

以上 3 种类型数都可以被进一步定义成无符号数：

```
unsigned char c;        // ranges from 0..255
unsigned int  i;        // ranges from 0..65,535
unsigned long l;        // ranges from 0..4,294,967,295
```

对于浮点运算有两种定义方法：

```
float      f;           // defines a 32 bit precision floating point
long double d;          // defines a 64 bit precision floating point variable
```

3.3.3 for 循环

现在回到计数器的例子。程序需要一个简单的整数变量来作为计数器，计数范围是从 0 到 5。因此一个 char 类型整数就可以满足要求：

```
char i;                 // declare i as an 8-bit integer with sign

i = 0;                  // init the index/counter

while ( i<5)
{
    // insert your code here...
    // it will be executed for i= 0, 1, 2, 3, 4

    i = i+1;            // increment
}
```

无论是加法计数还是减法计数，在每天的编程工作中都会用到不少。

在 C 语言中，还有第三种循环可以让通常的编码变得简单。那就是 for 循环。对于前一个例子，可以这样使用它：

```
for ( i=0; i<5; i=i+1)
{
    // insert your code here...
    // it will be executed for i=0, 1, 2, 3, 4
}
```

读者会发现 for 循环语法比较简洁，编写也相当容易，它同样也易于阅读和调试。关键字 for 后括号里的 3 条语句以分号隔开，和前面例子中的 3 个表达式完全一样：

- 初始化计数值；
- 使用逻辑表达式检查是否满足终止条件；
- 计数值更新，在本例中是自增 1。

可以将 for 循环看成是 while 循环的简写。实际上，逻辑表达式会先被计算，如果一开始就是“假”值，那么循环体括号里的代码就永远不会被执行。

现在来复习一下 C 语言中的其他一些有用的简写。C 语言为自增和自减操作保留了特殊的符号：

- ++　　自增，如 i++；就等于 i = i+1;
- --　　自减，如 i--；就等于 i = i-1;

第 4 章将会更多地介绍，不过到现在为止，符号已经够用了。

3.3.4　更多循环示例

下面是一些 for 循环和自增/自减运算的示例。

首先，考虑计数值从 0 增加到 4 的情况：

```
for ( i=0; i<5; i++)
{
    // insert your code here...
    // it will be executed for i= 0, 1, 2, 3, 4
}
```

然后，考虑计数值从 4 减到 0 的情况：

```
for ( i=4; i>=0; i--)
{
    // insert your code here...
    // it will be executed for i= 4, 3, 2, 1, 0
}
```

那么可以使用 for 循环作为（无限的）主程序循环吗？

当然可以，下面就是例子：

```
main()
{
    // 0. initialization code
    ...

    // 1. the main application loop
    for ( ; 1; )
    {
            ...
    }
} // main
```

如果读者喜欢，就可以采用这种形式。对于本书，从现在开始，还是继续采用 while 循环比较好（这是作者的旧习惯）。

3.3.5 数组

在开始下一项目的编程之前，需要复习一下C语言的最后一个特性：数组变量类型。数组其实就是在连续的内存区域里存放的一定数量的类型相同的元素。数组一旦被定义，每个元素都可以通过数组名和序号来访问。定义数组和定义单一变量一样简单——只要在变量名后的中括号里加入期望的元素数量就可以：

```
char c[10];    // declares c as an array of 10 x 8-bit integers
int i[10];     // declares i as an array of 10 x 16-bit integers
long l[10];    // declares l as an array of 10 x 32-bit integers
```

中括号还可以用来分配或访问数组的内容：

```
a = c[0];            // copy the value of the 1st element of c into a
c[1] = 123;          // assign the value 123 to the second element of c
i[2] = 12345;        // assign the value 12,345 to the third element of i
l[3] = 123* i[4];    // compute 123 x the value of the fifth element of i
```

> **注解** 在C语言里，大小为 N 的数组中的元素序号为 $0, 1, 2, \cdots, (N-1)$。当操作数组时，使用 for 循环具有较大的优势。

下面来看一个例子，定义一个有10个元素的数组，将数组中每个元素的值初始化为1：

```
int a[10]; // declare array of 10 integers: a[0], a[1], a[2]...
a[9]
int i;              // the loop index
for ( i=0; i<10; i++)
{
    a[ i] = 1;
}
```

3.3.6 新的演示程序

作为本章内容的结尾，将已经复习过的C语言应用于下面的项目中。该项目将一串连接到PORTA的LED显示器（已经连接到Explorer16演示板）点亮，并使其有节奏地闪烁，以显示一个较短的文字信息。

使用“Hello World”作为信息内容怎么样？又或者是更简短的“Hello”？

程序代码如下：

```
#include <p24fj128ga010.h>

// 1. define timing constant
#define SHORT_DELAY 100
#define LONG_DELAY 800

// 2. declare and initialize an array with the message bitmap
char bitmap[30] = {
    0b11111111,    // H
    0b00001000,
    0b00001000,
```

```
    0b11111111,
    0b00000000,
    0b00000000,
    0b11111111,    // E
    0b10001001,
    0b10001001,
    0b10000001,
    0b00000000,
    0b00000000,
    0b11111111,    // L
    0b10000000,
    0b10000000,
    0b10000000,
    0b00000000,
    0b00000000,
    0b11111111,    // L
    0b10000000,
    0b10000000,
    0b10000000,
    0b00000000,
    0b00000000,
    0b01111110,    // O
    0b10000001,
    0b10000001,
    0b01111110,
    0b00000000,
    0b00000000
    };

// 3. the main program
main()
{
    // 3.1 variable declarations
    int i;                  // i will serve as the index

    // 3.2 initialization
    TRISA =    0xff00;      // PORTA pins connected to LEDs are outputs
    T1CON =    0x8030;      // TMR1 on, prescale 1:256 Tclk/2

    // 3.3 the main loop
    while( 1)
    {
            // 3.3.1 display loop, hand moving to the right
            for( i=0; i<30; i++)
            {       // update the LEDs
                    PORTA = bitmap[i];
                    // short pause
                    TMR1 = 0;
                    while ( TMR1 < SHORT_DELAY)
                    {
                    }
            } // for i

            // 3.3.2 long pause, hand moving back to the left
```

```
            PORTA = 0;              // turn LEDs off
            // long pause
            TMR1 =  0;
            while ( TMR1 < LONG_DELAY)
            {
            }
        } // main loop
    } // main
```

在程序段 1 中，定义了一对时间常数，用于控制执行和调试时的灯光闪烁速度。

在程序段 2 中，定义了有 30 个元素的 8 位数组，每一元素都包含有 LED 显示序列配置。

提示：使用高亮显示，读者可以在页面上标注“1s”来查看浮现出的信息。

程序段 3 是主程序部分，开始是变量定义（程序段 3.1），然后是微控制器的初始化（程序段 3.2），最后是主循环体（程序段 3.3）。

主（while）循环由以下两部分组成。

- 包含 LED 显示序列，共 30 步，当板子从左到右扫描时会显示出来。for 循环用于访问每个数组中的元素。while 循环用于定时器 1 的计时。
- 包含每次扫描后的暂停。这可以使用 while 循环和定时器 1 产生延时的方法来实现。

3.3.7 使用逻辑分析器测试

为了测试程序，首先要使用 MPLAB SIM 软件仿真器和逻辑分析器窗口，如图 3-1 所示。

(1) 构建项目（使用合适的备忘录）。

(2) 打开“Logic Analyzer”（逻辑分析器）窗口。

(3) 单击 Channel（通道）按钮，将从 RA0 到 RA7 的所有 I/O 引脚连接到 LED 显示器组。

使用“MPLAB SIM Set-up”和“Logic Analyzer Set-up”列表可以保证读者操作全面。

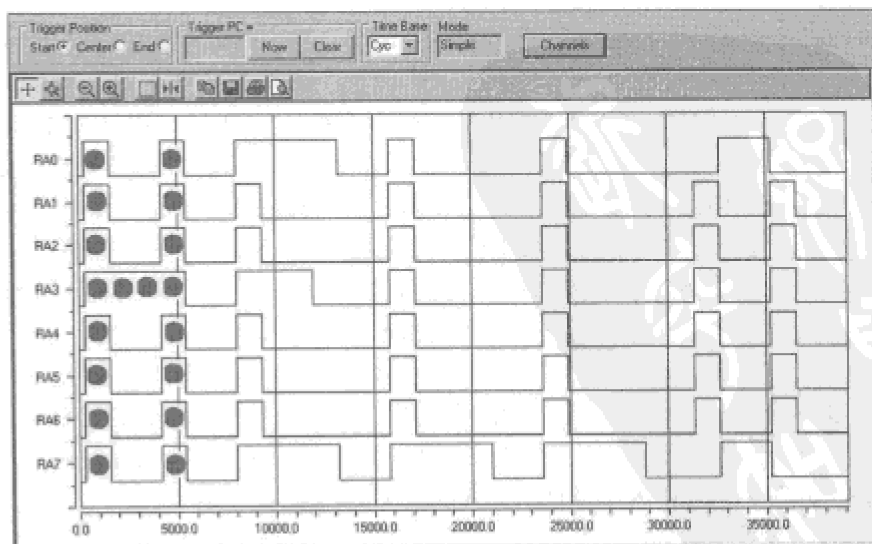


图 3-1 第一次扫描后的逻辑分析器窗口

接下来，建议读者回到编辑窗口，将鼠标指针置于程序段 3.3.2 中的第一条指令处，然后选择右键菜单的“Run to Cursor”选项。这样程序就会执行信息显示输出的整个程序段（程序段 3.3.1），并在长时暂停程序段前停止。当仿真器在光标所在行停止时，转到逻辑分析器窗口就可以查看输出波形。

为了使输出可视化，程序在显示序列的最初增加了一些小点来表示 LED 显示器已经打开。如果读者仔细观察引脚为高电平时对应的 LED 状态，就应该能够读出期望的信息。

3.3.8 使用 Explorer16 演示板

如果有真实的 Explorer16 演示板，读者将会感受到更大的乐趣。

(1) 使用“MPLAB ICD2 Set-up”列表，打开内部电路调试器。

(2) 使用“MPLAB ICD2 Device Configuration”来检验 Explorer16 演示板对应的设备配置位是否设置正确。

(3) 使用“MPLAB ICD2 Programming”列表对 PIC24 进行电路内编程。

如果设置成功，并把室内灯光调暗，当“摇晃”演示板的时候，读者就可以看到信息在闪动了。这个实验还不够完美。通过仿真器和逻辑分析器，还可以选择显示序列的某一部分或者让它“停”在屏幕上。在演示板上，读者可能会发现同步演示板的移动和 LED 序列是具有很大挑战性的。

现在把时间常数调整为理想的速度。经过一些实验，发现时间常数为 100 和 800 比较好，分别对应着短延时和长延时。不过读者使用的参数可能会有所不同。

3.4 飞后小结

本章复习了基本变量类型的定义，包括不同大小的整型数和浮点数，使用了数组的定义及其初始化来生成 LED 显示序列，使用 for 循环来回显 LED。

3.5 给汇编语言专家的提示

如果读者正在思考自增和自减操作符是不是和 C30 编译器的 inc 和 dec 汇编指令相同，那就大概对了。这里说的是“大概”，而不是“完全”，因为“++”和“--”操作符更为智能化。如果变量是整型，就像前面的例子那样，那它们就是一样的。但如果是用在指针（包含内存地址的变量类型）上，那么增量就是指针变量的字节数。例如，对于表示 16 位整型的指针，增量就是 2，对于表示 32 位长整型整数的指针，增量就是 4，如此类推。为了满足读者的好奇心，现切换到反汇编视图，观察 MPLAB C30 是如何根据状态选择最优汇编码的。

C 语言的循环有时会让人不知所措：应该在前面还是后面检查循环条件呢？是否应该使用 for 循环呢？实际上，在某些情况下，所采用的算法就决定了到底使用哪种循环，不过很多情况下用户都可以从几种循环里随意选择。最好选择一种让代码可读性更强的循环，如果没有什么大碍的话，可以在主循环体内选择自己喜欢的循环类型。

3.6 给 PIC 微控制器专家的提示

程序代码的紧凑程度以及执行效率，取决于目标微控制器的结构、最终的算术逻辑单元

(ALU) 以及操作数的访问宽度。在 PIC16 和 PIC18 的 8 位结构中，都是尽量采用字节大小的整数，而在 PIC24 中，16 位结构字大小的整数操作起来一样有效。对于习惯了 MPLAB C30 编译器的 16 位整型数的程序员来说，唯一的限制就是要考虑微控制器内宝贵的内部资源限制，在本章的例子中就是 RAM 内存。

3.7 给 C 语言专家的提示

即使 PIC24 微控制器有相当大的 RAM 存储器阵列，嵌入式控制的应用还是要考虑成本和空间限制等现实情况。如果要在 PC 或者工作站上使用 C 语言编程，可能就不会用到比 int 更小的整型数来作为循环计数器。好，现在又该想想了。当在应用中减少一个字节的设备需求时，在某些情况下就有可能选择小一点的 PIC24 微控制器，从而节省几块钱。如果需要大量制造（由成品的制造速度决定），这就能节省一大笔真金实银。换而言之，如果读者学会了把变量的数量严格控制在最小需求上，那么就能成为更好的嵌入式控制设计者，最终成为真正意义的工程师。

3.8 提示与技巧

这是本书的第 3 章，相信读者注意到了，这里已经是第三次指示读者把光标放置在程序的第一行，使用“Run To Cursor”命令（或者是设置断点）启动仿真器，而不是一直沿用更简单的单步执行。为什么要这么麻烦呢？为什么不在建立项目后直接开启动画模式呢？

就如已经不止一次提到的，这都是 c0 初始化代码的缘故。更进一步说，也是因为 MPLAB 把读者与低级的技术细节强制性地隔离开了。实际上，在单步执行的时候，MAPLAB 连光标（大的绿箭头）也不显示——真没有安全感。如果读者使用反汇编窗口，那么是追踪不到 c0 码的。不过 c0 码在开始的时候会做一些有趣的事情，而读者可能已经充满好奇心了。例如，在过去的练习中，定义了一个 bitmap[] 的数组，并使用指定的系列值对其初始化。在程序执行期间，数组（也就是数据结构）驻留在 RAM 中，因此编译器在程序开始后，需要指引 c0 初始化代码立即从 Flash 存储器的一个表格中复制数组的内容。

查看 c0 内部工作的唯一办法就是打开“View→Program Memory”（程序存储器窗口），选择 Symbolic（符号）模式（使用窗口底部的按钮），然后耐心地检查汇编代码。到处都是的标识可以为读者提供一点帮助。程序存储器窗口的第一行对应于 PIC24 的复位向量，而且总是带有转移到程序开始处的跳转指令：

```
0000        goto _reset
```

翻过几页之后就能看到中断向量表。最后，将会看到 _reset 标识。很快，读者就能辨认出代码的 4 个最主要的部分。

- 栈指针（w15）初始化

```
_reset mov.w #0x81e,w15
```

- 变量（RAM）初始化的子程序调用

```
rcall _data_init
```

- main()函数调用

```
call main
```

- 在程序末尾的软件复位指令

```
reset
```

希望以上的这些知识能够满足读者的好奇心。如果在以后的调试中，读者找不到光标的影子，也许就能在这里找到。有些情况可能会导致处理器的复位（一个小错误或者外部事件），这需要读者逐个检查 c0 初始化代码的核心部分。检查应急事件列表，有助于“飞行员”渡过难关，平安到家。

3.9 练习

(1) 改进显示/手动同步，在手动扫描之前等待按钮动作。

(2) 增加一个开关来控制扫描方向，在反向扫描中，让 LED 显示的顺序也反向。

3.10 推荐书目

- Rony, P., Larsen D. and Titus J., 1976
  The 8080A Bugbook, Microcomputer Interfacing and Programming
  Howard W. Sams & Co., Inc., Indianapolis, IN
  这本书将我引入了微处理器的世界，改变了我的生活。这本书中没有高难度的编程，只有基本汇编语言和硬件接口。（可惜这本书已进入博物馆，请参阅下面的网络链接）
- Shulman, S.(2003)
  Unlocking the Sky, Glenn Hammond Curtis and the Race to Invent the Airplane
  Harper Collins, New York, NY
  这本书用优美的语言介绍了早期飞行事业的革新奋斗。

3.11 网上链接

- http://www.bugbookcomputermuseum.com/BugBook-Titles.html
  这是“Bugbooks 博物馆”的链接——从 Intel 8080 微处理器问世到今天才刚刚 30 年，可是感觉好像过去了一个世纪。

第4章 数据类型

本章内容

- ▶关于优化
- ▶测试
- ▶走近长整型
- ▶长整型数据乘法的说明
- ▶双长整型数据的乘法
- ▶浮点型

人类的平衡感源自耳朵中的某种生理构造（内耳的迷路或者前庭器官）能及时向人们反馈重力和运动信息。但是，与鸟类不同，人天生是不会飞的。少许的离心加速度就足以蒙蔽我们。如果没有清晰的能见度（比如遭遇大雾、浓云或者在夜间飞行），那么我们很容易飞成愈来愈紧的螺旋状，直至撞向地面。为了克服人类在飞行上的缺点，飞行员不得不借助仪器去获知当前的飞行速度和飞行方向，以及（可能是最重要的）哪个方向是上。实际上，这意味着小鸟在飞行时大脑直接获得的信息要转变成数据的形式才能为飞行员所用。

对于一个飞行学员而言，在完成了最初的几次飞行之后，接下来要做的就是学习“正确的”飞行数据——比如最佳爬升速度、最佳滑行速度、起飞（旋转）速度、着陆速度等。大多数时候，这些数据可以从飞行员操作手册（POH）、飞机数据表以及相关的备忘录中查到。为了提高飞机的控制能力和保持飞行的稳定性，每个飞行员都会竭尽所能地听从这些数据的指挥。但是，那些经验丰富的特技表演飞行员和一些每年飞行上千小时的飞行员会告诉人们，只要精确地掌握了那些数据，飞行就会变得轻松自然。

嵌入式控制也是如此，程序员需要清楚地知道数据类型、相对的性能以及每种数据类型的优缺点。

4.1 飞行计划

本章将回顾 MPLAB C30 编译器提供的所有数值数据类型。读者将学习到编译器是如何为每种类型的数值变量分配存储空间的，以及在使用 MPLAB SIM 秒表（stopwatch）作为测量工具时如何评价执行算术运算的程序的相对效率。本章的内容将有助于读者为嵌入式控制应用选择“正确的”数据类型，有助于读者在性能与存储器资源、实时性限制与复杂度之间掌握好平衡。

4.2 飞前备忘录

本章将以特别的方式使用包括 MPLAB IDE、MPLAB C30 编译器、MPLAB SIM 仿真器等在内的软件。

使用“New Project Set-up”（新项目建立）列表创建一个新项目并命名为“Numbers”，创

建一个新的源文件并命名为“numbers.c”。

4.3 飞行

在回顾所有可用的数据类型之前，建议读者先浏览 MPLAB C30 的用户指南。在第 5 章，读者还会发现一个支持的整型数据表。

从表 4-1 可知，ANSI C 标准定义了 10 种不同的整数类型，包括：字符型（char）、整型（int）、短整型（short）、长整型（long）、双长整型（long long）等，每种数据类型又可以分为有符号数（默认）和无符号数两种。表 4-1 还给出了 MPLAB C30 编译器为每种数据类型分配的位数，以及每种数据类型的取值范围的最小值和最大值。

表 4-1 整型数据类型

类 型	位 数	最 小 值	最 大 值
char, signed char	8	–128	127
unsigned char	8	0	255
Short, signed short	16	–32 768	32 767
unsigned short	16	0	65 535
int, signed int	16	–32 768	32 767
unsigned int	16	0	65 535
long, signed long	32	-2^{31}	$2^{31}-1$
unsigned long	32	0	$2^{32}-1$
long long(1), signed long long(1)	64	-2^{63}	$2^{63}-1$
unsigned long long(1)	64	0	$2^{64}-1$

(1) ANSI-89 扩展

正如预想的那样，如果数据是有符号数，则必须使用一位来表示数据的符号，因此数据的取值范围也会减少一半。有趣的是，读者会发现 MPLAB C30 编译器将短整型和整型看作同一种类型，因为给它们分配的都是 16 位。PIC24 的算术逻辑单元（ALU）能够高效处理所有 8 位和 16 位的数据，以至于编译器仅使用少量高效的指令就能编译大多数的算术运算操作。长整型（long）数据是 32 位宽的，占用 4 个字节，双长整型（long long）数据（在 1989 年被列入 ANSI C 的扩展标准）占用 8 个字节。对长整型（long）数据的操作，编译器使用一系列内联的短指令来实现。因此，使用长整型数据会付出一定的性能代价，而使用双长整型数据的性能损失更大，这些都是必须预先考虑到的。

现在来看第一个整型数据的例子。首先是键入以下的代码段：

```
unsigned int i,j,k;

main ()
{
    i = 0x1234;    // assign an initial value to i
    j = 0x5678;    // assign an initial value to j
    k = i * j;     // perform the product and store the result in k
}
```

建立项目（Project→Build All 或 Ctrl+F10）之后，打开“Disassembly”（反汇编）窗口（“View→Disassembly Listing”），查看编译器产生的代码。即使不熟悉 PIC24 详细的指令集，读者也能够看得懂这两条赋值语句。它们首先将立即数传送给寄存器 w0，再将寄存器 w0 的值传送到存储器中为变量 i 预留的地址，从而完成对变量 i 的赋值。再接着对变量 j 赋值，方法相同。

```
                i = 1234;
204D20      mov.w #0x4d2,0x0000              // move literal value to W0
884290      mov.w 0x0000,0x0852              // move data from W0 to i

                j = 5678;
2162E0      mov.w #0x162e,0x0000             // move literal value to W0
8842A0      mov.w 0x0000,0x0854              // move data from W0 to j

                k = i * j;
804291      mov.w 0x0852,0x0002              // move data from i to W1
8042A0      mov.w 0x0854,0x0000              // move data from j to W0
B98800      mul.ss 0x0002,0x0000,0x0000
8842B0      mov.w 0x0000,0x0856              // move result  to k
```

乘法运算的实现是将内存中为变量 i 和变量 j 预留的地址中的值分别传送给寄存器 w0 和 w1，然后执行一条简单的 mul（乘法）指令。乘法运算的结果在寄存器 w0 中，将此结果传送回到存储器中为变量 k 预留的地址。非常简单易懂！

4.3.1 关于优化

读者会发现所编译的程序中有些部分是冗余的。比如说，在执行乘法指令之前将变量 j 的值传送给寄存器 w0，而实际上变量 j 的值已经在寄存器 w0 中。难道编译器没察觉出这步操作是不必要的吗？

事实上，编译器并不能对每件事都掌握得很清楚——它的任务是产生“安全的”代码，它需要避免（至少在最初的时候）任何的假设，并且使用标准的指令。这样，后续当选用了合适的优化方法时，仅用很短的时间就可以删去其中的冗余代码。但是，在项目的开发和调试阶段，最好不要使用任何代码优化，因为优化方案可能会修改正在被分析的代码结构，并给单步执行和断点设置带来问题。在本书的其余章节，将会一如既往地避免使用任何编译器优化方案，但是不管怎样都会保证程序具有预期的性能水平。

4.3.2 测试

对于测试代码，读者可以使用“Disassembly Listing”（反汇编列表）窗口的仿真器，单步执行每一条汇编指令。或者，读者也可以使用编辑窗口中的 C 源程序，单步执行其中的每一条 C 语句。在这两种情况下，读者可以完成以下配置。

(1) 将光标置于第一行对第一个变量进行初始化的语句，然后使用“Run To Cursor”命令来初始化程序，并在光标到达第一个需要被观察的指令前停止执行。

(2) 打开“Watch”（监视）窗口（“View→Watch”），在“SFR selection box”（SFR 选择框）中选择“WREG0”，然后单击“Add SFR”按钮。

(3) 对寄存器 WREG1 重复同样的操作。

(4) 从“symbol selection box”（符号选择框）中选择“i”，然后单击“Add Symbol”按钮。

(5) 对变量 j 和变量 k 重复同样操作。

(6) 使用“Step Over”功能执行光标以下的代码行，观察“Watch window”中寄存器和变量的变化。正如前面提到的，如果“Watch window”中有一个变量的值改变，那么该变量将会以红色突出显示。

如果需要重复测试，则使用“Reset”（“Debugger→Reset→Processor Reset”）命令，同时再次将光标置于代码的第一行，接下来便是执行一个新的“Run To Cursor”命令。

4.3.3 走近长整型

现在，读者只需要改变代码的第一行，就可以让整个程序所指向的操作数变成长整型变量。

```
unsigned long i,j,k;

main ()
{
        i = 0x1234;    // assign an initial value to i
        j = 0x5678;    // assign an initial value to j
        k = i * j;     // perform the product and store the result in k

}
```

重新构建项目，并再次切换到“Disassembly Listing”（反汇编列表）窗口（如果“editor”（编辑）窗口已最大化且“Disassembly Listing”（反汇编列表）窗口没有关闭，读者可以使用 Ctrl+Tab 命令在编辑窗口和反汇编列表窗口之间进行快速的切换），读者将会发现，新程序编译生成的代码长度较前一个程序增加了不少。对于新的程序，尽管其初始化仍然简明易懂，但是执行乘法操作时就需要更多的指令。

```
              k = i * j;
8042C1        mov.w 0x0858,0x0002
8042E0        mov.w 0x085c,0x0000
B80A00        mul.uu 0x0002,0x0000,0x0008
8042C1        mov.w 0x0858,0x0002
8042F0        mov.w 0x085e,0x0000
B98800        mul.ss 0x0002,0x0000,0x0000
780105        mov.w 0x000a,0x0004
410100        add.w 0x0004,0x0000,0x0004
8042E1        mov.w 0x085c,0x0002
8042D0        mov.w 0x085a,0x0000
B98800        mul.ss 0x0002,0x0000,0x0000
410100        add.w 0x0004,0x0000,0x0004
780282        mov.w 0x0004,0x000a
884304        mov.w 0x0008,0x0860
884315        mov.w 0x000a,0x0862
```

PIC24 的算术逻辑单元（ALU）每次只能处理 16 位的操作数，因此 32 位数的乘法实际上是通过执行 16 位数的乘法运算和加法运算来实现的。编译器运用类似于在小学中学到的方法来实现运算的执行顺序，不过并不是每次只对一位数进行操作，而是每次对 16 位数进行操作。

4.3.4 长整型数据乘法说明

在实践中，使用 16 位指令实现 32 位乘法运算，需要执行 4 次乘法运算和 2 次加法运算。但是读者会发现编译器实际上只插入了 3 条乘法指令。这是怎么回事呢？

事实上，两个双长整型数据（每个数据为 32 位）相乘将会产生 64 位的结果。但是在上面的例子中，已经指定将乘法运算的结果存放在长整型变量中，即将结果限制为 32 位。虽然这样的做法有可能导致溢出发生，但是这里已允许编译器忽略所得结果中的最高有效位。由于已知最高有效位不会丢失，编译器便省略了第四个乘法运算，从而在某种程度上达到了优化代码的效果。

4.3.5 双长整型数据的乘法

将变量声明部分改为对双长整型变量的声明是非常简单的：

```
unsigned long long i,j,k;

main ()
{
    i = 0x1234;    // assign an initial value to i
    j = 0x5678;    // assign an initial value to j
    k = i * j;     // perform the product and store the result in k

}
```

重新编译并且观察“Disassembly Listing”（反汇编列表）窗口中的结果，将会发现编译器选择了另外一种方法来编译此段代码。这里不再使用长长的一串内联指令，而是仅使用几条指令将数据传送给预定义的寄存器，然后调用一个子程序。子程序在“Disassembly Listing”（反汇编列表）中的主函数代码的后面，并且使用一个注释行来说明此子程序属于库函数“muldi3.c”。事实上，该程序的源文件已包含在 C30 编译器的文件中，读者可以在电脑上 C 编译器的安装路径下的子目录“src/libm/src/”中找到它。

在这里选用子程序，实际上是编译器的一种妥协处理。因为调用子程序意味着要加入一些额外的指令，并且会占用栈中额外的空间；不过这样做可以减少程序中每次乘法运算（双长整型数据之间的乘法运算）需要使用的指令数，从而保护了代码空间。

4.3.6 浮点型

除了整型数据，C30 编译器还提供一些其他的数据类型用来表示小数——浮点型数据。基于两种不同层次的处理方法，共有 3 种浮点型数据：浮点型（float），双精度浮点型（double）和长双精度浮点型（long double），如表 4-2 所示。

表 4-2 浮点数据类型

类 型	位	E 最小值	E 最大值	N 最小值	N 最大值
浮点型	32	−126	127	2^{-126}	2^{128}
双精度浮点型 [1]	32	−126	127	2^{-126}	2^{128}
长双精度浮点型	64	−1 022	1 023	2^{-1022}	2^{1024}

E＝指数值。
N＝标准值（近似值）。
(1) 使用-fno-short-double 时，双精度浮点型和长双精度浮点型是等价的。

注意，在默认情况下，C30 编译器根据 IEEE754 标准定义的单精度浮点型格式，对浮点型和双精度浮点型这两种数据类型分配了相同的位数。只有长双精度浮点型数据类型才是真正的双精度 IEEE754 浮点型。

4.4　给 C 语言专家的提示

我认为，MPLAB C30 的设计者特意使用浮点型设置来使得嵌入式控制目标应用中烦琐的数学计算变得更为精简高效。大多数的运算法则和库都是基于个人电脑和工作站的性能和资源而设计的，从而尽可能地使用双精度浮点型运算以达到最高的精度。在嵌入式应用中，往往会以降低一定的精度为代价来换取程序更好的实时响应效果。

如果需要，可以局部地或者全部地将双精度浮点型转换为长双精度浮点型，要完成这个工作可使用特殊的编译器选项（打开“Project→Build Options→Project”对话框，检查“Use alternate Setting check box”，添加“-fno-short-double”到“edit box”底部）。

由于 PIC24 没有硬件浮点数单元（FPU），所以所有浮点数的操作都必须由编译器使用浮点算术运算库来编码处理，而该算术运算库的大小和复杂度比任何的整数算术运算库都要大/高很多。因此，若要使用浮点型数据，就应该考虑到性能的损失。当然，如果程序要求使用分数，C30 编译器也能轻松地处理。

下面将前面的示例程序修改为使用浮点型变量的情况：

```
float i,j,k;

main ()
{
    i = 12.34;      // assign an initial value to i
    j = 56.78;      // assign an initial value to j
    k = i * j;      // perform the product and store the result in k

}
```

重新编译此程序，并且检查“Disassembly Listing”（反汇编列表）窗口，读者将会发现编译器选择了使用子程序（subroutine）而不是内联代码（inline code）。

再次修改示例程序，使用双精度浮点型数据，产生的结果与刚才相似。只有赋值语句受到了影响，读者能看到的只是一个子程序的调用。

由于 C 编译器对每一种数据类型都可以灵活地使用，往往造成程序员在编程时选择最大的整数或者浮点型数据以确保程序运行的安全性，避免潜在的上溢出和下溢出风险。然而，数据类型的正确选择对于嵌入式控制的性能与资源的平衡非常重要。为了选择一个最合适的精度，读者需要了解程序在不同精度下的性能。

性能评测

下面将会使用到目前为止介绍过的各种仿真工具来评测 C30 编译器所使用的算术运算库（整型和浮点型）的性能。首先，使用软件仿真器（MPLAB SIM）的内置秒表工具，其代码如下：

```
//
//   Numbers
//

int   i1, i2, i3;
long  l1, l2, l3;
long long ll1, ll2, ll3;
float f1,f2, f3;
long double d1, d2, d3;

main ()
{
    i1 = 1234;     // testing integers (16-bit)
    i2 = 5678;
    i3= i1 * i2;   // 1. int multiplication

    l1 = 1234;     // testing long integers (32-bit)
    l2 = 5678;
    l3= l1 * l2;   // 2. long multiplication

    ll1 = 1234;    // testing long long integers (64-bit)
    ll2 = 5678;
    ll3= ll1 * ll2;      // 3.

    f1 = 12.34;    // testing single precision (32-bit) floating point
    f2 = 56.78;
    f3= f1 * f2;   // 4. single precision multiplication

    d1 = 12.34;    // testing double precision (64-bit) floating point
    d2 = 56.78;
    d3= d1 * d2;   // 5. double precision multiplication

}
```

编译并连接此段代码之后，在“编辑”（editor）窗口中将光标（cursor）置于包含第一个整型数据乘法运算（程序段 1）的语句处，执行“Run To Cursor”以定位程序计数器的位置。打开“秒表”（Stopwatch）窗口（“Debugger→Stopwatch”）并且根据个人喜好设置此窗口的位置（就个人而言，本人喜欢将秒表窗口置于屏幕的下端，这样既不会覆盖编辑窗口，同时又能够随时查看并使用此窗口）。

将秒表的定时器清零，执行“Step-Over”指令（“Debugger→Step-Over”，或者按下 F8）。仿真器每更新完一次秒表（Stopwatch）窗口，读者就可以手动记录下完成一次整型数据操作所需的时间。仿真器提供的时间格式为一个循环计数值及其对应的毫秒值，该毫秒值可由循环计数值和仿真器时钟频率相乘得到。其中仿真器时钟频率为“调试器设置”（Debugger Settings）中指定的参数（“Debugger→Settings→Osc/Trace”）。

继续测试。将光标置于第二个乘法运算语句处（程序段 2），并且执行新的“Run To Cursor”指令，或者可以一步一步地向下执行，直至第二个乘法运算语句处。将秒表清零，执行“Step-Over”指令并记录时间。继续测试直至 5 种类型数据的乘法运算全部测试完毕。

在表 4-3 中，第一列数据为测试结果（循环计数值），其他列数据为数据类型的相对性能比

值（由每行循环计数值除以对应列数据类型的循环计数值可以得到）。如果读者得到的测试结果与表中给出的不一致，也不必惊慌，因为测量往往会受到很多因素的影响。以后版本的编译器也许能够使用更高效率的库或者在测试的时候可以使用最优化设计。

表 4-3 使用 MPLAB C30 1.30 版本的相对性能测试结果（禁止使用所有的优化方法）

乘法运算测试	循环计数值	相对性能		
		int	long	Float
整型	4	1	—	—
长整型	15	3.75	1	—
双长整型	99	24.75	6.6	—
单精度浮点型	121	30	8	1
双精度浮点型	317	79	21	2.6

请读者谨记，同真实的性能测试平台相比，这种测试没有严格的测试条件，而只是大体上的算术运算比较，即一种数据类型与其他类型数据对性能影响的程度区别而已。作者期望的是得到各类型数据的运算速度的排序，基于这个目标，上面的表格已经给出了有趣的答案。

正如所期望的那样，16 位的操作是最快的。长整型（32 位）乘法运算慢了 4 倍，而双长整型（64 位）乘法运算则慢了一个数量级。同时，单精度浮点型运算可能会比整型运算花费更多的时间。32 位的整数相乘只比 16 位的整数相乘慢 4 倍；而 32 位的浮点型数据相乘却比 16 位的整数相乘慢了 30 多倍，即较相应的 32 位整数相乘而言慢了 8 倍或者说是降了将近一个数量级。使用双精度浮点型（64 位）数据的循环计数值仅仅是 16 位整型数据的两倍，很显然这说明了编译器使用的双精度浮点库比 64 位整数库更为高效。

那么，应该在何时使用浮点型数据，又在何时使用整型数据进行算术运算呢？

除了上面的数据，也可以从以上所介绍过的知识中提取出以下几点规则。

(1) 尽量使用整型数据（即在不要求使用小数或分数时，或者在算法可以改写成整数运算的情况下）。

(2) 在确保不会产生上溢出和下溢出的情况下使用最小的整数类型。

(3) 当必须使用浮点型数据时（在要求使用小数时），已考虑到编译后的程序性能将会降低一个数量级。

(4) 使用双精度浮点数据类型会将性能降低一半。

请记住，浮点型数据提供了最大的取值范围，同时也会带来近似值。因此，建议读者不要使用浮点型数据进行财务计算，代替的是使用长整型或双长整型数据，并且所有的运算都以美分（而不是美元或者小数）为计量单位。

4.5 飞后小结

本章不仅介绍了 PIC24 微控制器可支持的数据类型及其所占用的存储器空间，还分析了这些数据类型如何影响程序的编译结果——代码大小和执行速度。使用 MPLAB SIM 仿真器中的秒表功能测定每次执行一串代码所需的指令周期数（再据此可计算出所需的时间）。之所以将这

些知识汇集在一起来介绍，是因为希望能够为读者在以后的嵌入式控制应用中对平衡精度和性能提供帮助。

4.6 给汇编语言专家的提示

那些试图在程序中处理浮点型数据的少数勇敢的汇编语言专家们，对于使用 C 编译器所带来的极大简化将是非常欣喜和感激的。不管是单精度运算还是双精度运算，都像整数运算一样容易编译。

由于 C 编译器隐藏运算的实现细节，同时有些运算是不太直观可读的，所以哪怕是使用整数，有时候也会有失控的感觉。以下是一些可能催生疑虑的数据类型转换操作和字节操作的例子。

(1) 将整数转换为更小或更大的数。

(2) 提取或设置 16 位数据类型的最高或者最低有效字节。

(3) 提取或设置整型变量的某一位。

C 语言提供了很方便的方法来解决这些问题，比如使用下面的隐式类型转换：

```
int       i;     // 16-bit
long      l;     // 32-bit
l = i;           // the value of i is transferred into the two LSB of l
                 // the two MSB of l are cleared
```

在有些情况下可能需要使用显式类型转换（又称为“强制类型转换”），否则编译器可能会产生错误，请看下面的例子：

```
int       i;     // 16-bit
long      l;     // 32-bit
i = (int) l;     // (int) is a type cast that results in the two MSB of l
                 // to be discarded as l is treated as a 16-bit value
```

位字段用于宽度小于一个字节的整数类型转换。由于 MPLAB C30 编译器对位字段的操作非常高效，使得位操作指令使用起来极其方便。PIC24 库文件包含有大量的用于操作外部寄存器和内部特殊功能寄存器的控制位的位字段定义例子。

以下是从本章项目所用到的 include 文件中摘取的程序片段，其中定义了定时器 Timer1 的控制寄存器 T1CON，每个独立控制位属于 T1CONbits 结构体。

```
extern unsigned int  T1CON;
extern  union {
  struct {
    unsigned :1;
    unsigned TCS:1;
    unsigned TSYNC:1;
    unsigned :1;
    unsigned TCKPS0:1;
    unsigned TCKPS1:1;
    unsigned TGATE:1;
    unsigned :6;
    unsigned TSIDL:1;
    unsigned :1;
```

```
    unsigned TON:1;
  };
  struct {
    unsigned :4;
    unsigned TCKPS:2;
  };
} T1CONbits;
```

4.7 给 PIC 微控制器专家的提示

熟悉各种 8 位 PIC 微控制器及其编译器的 PIC 微控制器用户将会发现，不管是整数运算性能还是浮点型数据运算性能，PIC24 微控制器都有相当大的提高。PIC24 微控制器使用的 16 位算术逻辑单元（ALU）提供了一个非常好的优点：每个周期可以处理的位数是 8 位微控制器的两倍。为提高性能做出最大贡献的是 8 个工作寄存器，它们使得主要的算法程序和数值算法的编译更为高效。

4.8 提示与技巧

4.8.1 函数库

MPLAB C30 编译器提供了一些标准 ANSI C 函数库，包括以下几种。

- "limits.h"，它提供了许多的定义使用限制的宏指令，比如字符型数据的位数（CHAR_BIT）或者整型数据的最大值（INT_MAX）。
- "float.h"，它提供了用于浮点数据类型的使用限制定义，比如用于单精度浮点型变量的最大指数值（FLT_MAX_EXP）。
- "math.h"，提供了三角函数、取整函数、对数函数、指数函数等。

4.8.2 复数数据类型

MPLAB C30 编译器支持复数（complex）数据类型，作为整型和浮点型数据类型的拓展。下面的例子是对单精度浮点型变量的声明：

```
__complex__ float z;
```

注意，在关键字 complex 的前后使用了两条下划线。

对于上面已经定义了的变量 z，可以使用_real_ z 和_imag_ z 两条语法分别对它的实部和虚部进行单独的访问。

类似地，以下是对 16 位整型变量的声明例子：

```
__complex__ int x;
```

通过将 i 或者 j 作为后缀，可以生成一个复数常量，如下所示：

```
x = 2 + 3j;
z = 2.0f + 3.0fj;
```

所有标准算术运算（即+、–、*和/）对复数数据类型都适用。此外，"~" 运算符可用来求取复数的共轭。

在某些类型的应用中，复数类型是相当方便的，它使得代码更加易读，并且能帮助读者避免微小的错误。但是，MPLAB IDE 在调试程序时只支持复数变量的一部分，在“Watch”窗口（监视）和鼠标跟随功能（mouse-over function）中只允许处理复数的实部。

4.9 练习

(1) 编制程序。使用定时器 2（Timer2）作为秒表来测量实时性能。如果 Timer2 的宽度不够，则使用预分频器（prescaler）（这将损失最低有效字节），或者在新的 32 位定时器模式中，将 Timer2 和 Timer3 联合使用。

(2) 测试不同数据类型的相对性能。

(3) 测试三角函数相对于标准算术运算的性能。

(4) 测试复数类型数据的乘法的相对性能。

4.10 推荐书目

- Gahlinger, P.M.(2000)
  The Cockpit, a Flight of Escape and Discovery
  Sagebrush Press, Salt Lake City, UT
  这是一趟周游世界的有趣旅行，请跟随作者去追寻……他的思想。
  驾驶舱里的每个仪器都将成为一段回忆，并将开启新的一章。

4.11 网上链接

- http://en.wikipedia.org/wiki/Taylor_series
  如果读者对 C 编译器如何逼近数学库中的一些函数感兴趣，可以登录这个网址查看。



第 5 章　中　断

本章内容

- ▶中断嵌套
- ▶陷阱
- ▶Timer1 中断的模板和示例
- ▶Timer1 应用实例
- ▶Timer1 中断的测试
- ▶二级振荡器
- ▶实时时钟日历（RTCC）
- ▶多个中断的管理

每个飞行员都会被教导要紧盯地平线，搜集关于飞行方向和位置的任何视觉信息和其他飞机的情况。但是，飞行员还必须观察飞机上的仪器来验证飞行的高度和速度，并密切注视飞行地图。偶尔还需要飞行员提供输入，更重要的是有些仪器的输入操作会比较频繁，这取决于飞行阶段和其他的飞行状态。换而言之，飞行员要懂得处理多个任务，给每个仪器分配适当的优先级，最优化地使用，以便永远赶在机器的前面。

鉴于嵌入式控制领域中效率、尺寸和成本等因素，以最大体积实现的最小应用通常都承受不起“奢华”的多任务操作系统。当有多个任务需求时，应采用中断机制来取代“分散注意力”的处理方式。

5.1　飞行计划

本章将介绍 MPLAB C30 编译器是如何轻松地管理 PIC24 微控制器的中断机制的。在简单回顾 C 语言的一些扩展知识和实践性的注意事项后，将使用一个简单的例子来说明如何使用二级（低频）振荡器以维持实时时钟信号。

5.2　飞前备忘录

本章只用到软件工具，包括 MPLAB IDE 集成开发环境、MPLAB C30 编译器和 MPLAB SIM 仿真器。

使用“New Project Set-up”列表创建名为“Interrupts”（中断）的新项目，并相似地创建名为“interrupts.c”的源文件。

5.3　飞行

中断是指需要 CPU 快速响应的内部或外部事件。PIC24 结构提供了一个丰富的中断系统，可以处理最多 118 个不同的中断源。每一个中断源都有唯一的代码段，即中断服务子程序(ISR)，用来提供期望的响应操作。与其相对应的指针，被称作“中断向量”。中断与主程序的执行流完全是异步的。中断可以在任何时间或者以任意顺序被触发。快速的中断响应，可以让系统对触

发事件作出迅速的反应，并立即返回，继续执行主程序。因此，嵌入式应用设计要最小化中断延时，即使得从触发事件到执行中断服务子程序（ISR）的第一条指令之间的间隔时间最小。在PIC24 结构中，延时时间不仅非常短，而且对于每一个既定的中断源都是固定的——内部事件仅需 3 个指令周期而外部事件仅需 4 个指令周期。这一特性使得 PIC24 的中断管理要优胜于其他的微控制器结构。

MPLAB C30 编译器为管理复杂的中断系统提供了一些语言扩展内容。PIC24 将所有的中断向量都保存在一个大的中断向量表（IVT）中，MPLAB C30 可以自动将中断向量与“特殊的”用户自定义函数关联起来。用户自定义的函数需要满足以下要求。

- 不返回任何数值（使用类型为 void）。
- 不传递任何参数到函数（使用参数为 void）。
- 不能被其他函数直接调用。
- 不能调用其他函数。

前三条要求显然是由中断机制的本质决定的——既然是由外部事件触发的中断，就没有先行的函数调用，因此不能参数传递或者返回值。最后一条是出于对程序执行效率的考虑而提出的建议，希望读者能记住。

下面的例子说明了一个函数与 Timer1 中断向量相关联的语法：

```
void __attribute__ (( interrupt)) _T1Interrupt ( void)
{
  // interrupt service routine code here...

} // _InterruptVector
```

函数名_T1Interrupt 并不是随意选择的，它是 PIC24 中断向量表中预先定义的 Timer1 中断标志符，（在数据表中已有定义）并且在连接器脚本里需要加载代码，对于本项目，加载的是“.gld”文件。

在这里和其他许多环境中，C30 编译器用到的_attribute_(())机制是用于说明特殊功能的 C 语言扩展功能。个人认为，这种语法又长又难读。推荐使用每个 PIC24 的 include 头文件（“.h”）里的宏定义，这可以大大提高代码的可读性。在下面的例子中，使用宏_ISR 实现的功能与前面的代码段相同：

```
void _ISR _T1Interrupt (void)
{
  // interrupt service routine code here...

} // _InterruptVector
```

表 5-1 摘自 PIC24FJ128G010 系列的数据表，从表中可知哪些事件可引起触发中断。对于 PIC24FJ128GA010 来说，可以触发中断的外部事件有：

- 5 个带有电平触发检测功能的外部引脚；
- 22 个连接到变化通知模块的外部引脚；
- 5 个输入捕捉模块；
- 5 个输出比较模块；

- ❑ 2个串行接口（UART）；
- ❑ 4个同步串行接口（SPI 和 I^2C）；
- ❑ 并行主端口。

而内部事件有：

- ❑ 5个16位的定时器；
- ❑ 1个模数转换器；
- ❑ 1个模拟比较器模块；
- ❑ 1个实时时钟及日历；
- ❑ 1个CRC发生器。

以上的每一种中断源都可以引起多种不同的中断。例如，外部串行接口（UART）可以引起以下3种中断。

- ❑ 当接收到新数据，并在接收缓冲器中等候处理时。
- ❑ 当传输缓冲器中的数据被发送出，缓冲器变为空，并准备就绪可传输新的数据时。
- ❑ 当发生错误并需要重建通信时。

每个中断源都有5个相关的控制位，它们位于不同的特殊功能寄存器里（如表5-1所示）。

- ❑ 中断使能位（通常用后缀-IE表示）：
  - ■ 为0时，指定的触发事件将被禁止产生中断；
  - ■ 为1时，允许中断被处理。

表 5-1　PIC24FJ128GA010 系列实现的中断向量

中　断　源	向量编号	IVT 地址	AIVT 地址	中断位地址		
				标志位	使能位	优先级
ADC1 转换完成	13	00002Eh	00012Eh	IFS0<13>	IEC0<13>	IPC3<6 : 4>
比较事件	18	000038h	000138h	IFS1<2>	IEC1<2>	IPC4<10 : 8>
CRC 发生器	67	00009Ah	00019Ah	IFS4<3>	IEC4<3>	IPC16<14 : 12>
外部中断 0	0	000014h	000114h	IFS0<0>	IEC0<0>	IPC0<2 : 0>
外部中断 1	20	00003Ch	00013Ch	IFS1<4>	IEC1<4>	IPC5<2 : 0>
外部中断 2	29	00004Eh	00014Eh	IFS1<13>	IEC1<13>	IPC7<6 : 4>
外部中断 3	53	00007Eh	00017Eh	IFS3<5>	IEC3<5>	IPC13<6 : 4>
外部中断 4	54	000080h	000180h	IFS3<6>	IEC3<6>	IPC13<10 : 8>
I2C1 主事件	17	000036h	000136h	IFS1<1>	IEC1<1>	IPC4<6 : 4>
I2C1 从事件	16	000034h	000034h	IFS1<0>	IEC1<0>	IPC4<2 : 0>
I2C2 主事件	50	000078h	000178h	IFS3<2>	IEC3<2>	IPC12<10 : 8>
I2C2 从事件	49	000076h	000176h	IFS3<1>	IEC3<1>	IPC12<6 : 4>
输入捕捉 1	1	000016h	000116h	IFS0<1>	IEC0<1>	IPC0<6 : 4>
输入捕捉 2	5	00001Eh	00011Eh	IFS0<5>	IEC0<5>	IPC1<6 : 4>
输入捕捉 3	37	00005Eh	00015Eh	IFS2<5>	IEC2<5>	IPC9<6 : 4>
输入捕捉 4	38	000060h	000160h	IFS2<6>	IEC2<6>	IPC9<10 : 8>
输入捕捉 5	39	000062h	000162h	IFS2<7>	IEC2<7>	IPC9<14 : 12>

（续）

中 断 源	向量编号	IVT 地址	AIVT 地址	中断位地址		
				标志位	使能位	优先级
输入变化通知	19	00003Ah	00013Ah	IFS1<3>	IEC1<3>	IPC4<14：12>
输出比较 1	2	000018h	000118h	IFS0<2>	IEC0<2>	IPC0<10：8>
输出比较 2	6	000020h	000120h	IFS0<6>	IEC0<6>	IPC1<10：8>
输出比较 3	25	000046h	000146h	IFS1<9>	IEC1<9>	IPC6<6：4>
输出比较 4	26	000048h	000148h	IFS1<10>	IEC1<10>	IPC6<10：8>
输出比较 5	41	000066h	000166h	IFS2<9>	IEC2<9>	IPC10<6：4>
并行主端口	45	00006Eh	00016Eh	IFS2<13>	IEC2<13>	IPC11<6：4>
实时时钟/日历	62	000090h	000190h	IFS3<14>	IEC3<13>	IPC15<10：8>
SPI1 错误	9	000026h	000126h	IFS0<9>	IEC0<9>	IPC2<6：4>
SPI1 事件	10	000028h	000128h	IFS0<10>	IEC0<10>	IPC2<10：8>
SPI2 错误	32	000054h	000154h	IFS2<0>	IEC0<0>	IPC8<2：0>
SPI2 事件	33	000056h	000156h	IFS2<1>	IEC2<1>	IPC8<6：4>
Timer1	3	00001Ah	00011Ah	IFS0<3>	IEC0<3>	IPC0<14：12>
Timer2	7	000022h	000122h	IFS0<7>	IEC0<7>	IPC1<14：12>
Timer3	8	000024h	000124h	IFS0<8>	IEC0<8>	IPC2<2：0>
Timer4	27	00004Ah	00014Ah	IFS1<11>	IEC1<11>	IPC6<14：12>
Timer5	28	00004Ch	00014Ch	IFS1<12>	IEC1<12>	IPC7<2：0>
UART1 错误	65	000096h	000196h	IFS4<1>	IEC4<1>	IPC16<6：4>
UART1 接收器	11	00002Ah	00012Ah	IFS0<11>	IEC0<11>	IPC2<14：12>
UART1 发送器	12	00002Ch	00012Ch	IFS0<12>	IEC0<12>	IPC3<2：0>
UART2 错误	66	000098h	000198h	IFS4<2>	IEC4<2>	IPC16<10：8>
UART2 接收器	30	000050h	000150h	IFS1<14>	IEC1<14>	IPC7<10：8>
UART2 发送器	31	000052h	000152h	IFS1<15>	IEC1<15>	IPC7<14：12>

在通电时，所有中断都是默认为禁止的。

- ❑ 中断标志位（通常用后缀-IF 表示）。这一位在每次特定的触发事件发生时都会置 1，与使能位的状态无关。注意，中断标志位一旦被置 1，那么它就需要用户手动清零。换而言之，在退出中断服务子程序前必须将该位清 0，否则，相同的中断服务子程序将立即被再次调用。
- ❑ 优先级级别标志位（通常用后缀-IP 表示）。中断优先级共有 7 个级别。如果两个中断同时出现，系统会先执行优先级高的中断服务子程序。每个中断源都有三位数用来表示其优先级。在任何时候，PIC24 执行的中断优先级值都会保存在 SR 寄存器内，并使用 IPL0..IPL2 分别表示优先级的三位数。如果中断的优先级低于目前的 IPL 值，那么该中断就会被忽略。在通电时，所有中断源的默认优先级是 4，处理器的默认优先级是 0。

在分配的优先级下，不同的中断源在 IVT 表中还有一个固定的发生顺序，这被称作相对（默认）优先级。

5.3.1　中断嵌套

中断是可以嵌套的，因此低优先级的中断服务子程序会被高优先级的中断服务子程序打断。这个过程可由 PIC24 的 INTCON1 寄存器的 NSTDIS 位来控制。

当 NSTDIS 位为 1 时，如果接收到中断，那么处理器的优先级（IPL）就会被置为最高级（7），并且与分配给事件的特定中断优先级无关。这样，在当前中断结束前就阻止了新中断的响应。换而言之，当 NSTDIS 位为 1 时，如果多个中断同时发生，则每个中断的优先级都只是用来解决冲突的，使所有中断都能按顺序执行。

5.3.2　陷阱

在 IVT 表的最前面有 8 个辅助的向量，它们是用于捕捉特殊错误条件的，如选择 CPU 振荡器失败、地址错误（使用字访问奇地址）、栈下溢出或者除数为 0（数学错误），如表 5-2 所示。

表 5-2　陷阱向量信息

向量编号	IVT 地址	陷阱源
0	000004h	保留
1	000006h	振荡器错误
2	000008h	地址错误
3	00000Ah	栈错误
4	00000Ch	数学错误
5	00000Eh	保留
6	000010h	保留
7	000012h	保留

由于以上的错误将会给程序运行造成致命的后果，所以它们被分配有固定的优先级，并且要高于分配给其他中断的 7 个基本优先级。也就是说，这些中断不能被随意屏蔽（或者被 NSTDIS 机制延时），它们给应用程序提供了更高级的保护。MPLAB C30 编译器使用一个可以引起处理器复位的默认子程序来关联所有的陷阱向量。用户可以使用适用于所有派生中断服务子程序的相同方法来改变这种默认设置。

5.3.3　Timer1 中断的模板和示例

下面的内容可能看起来很复杂，不过读者很快就能发现，按照几个简单的指引，马上就能把它付诸实际。首先要生成一个模板，在以后的实践例子中还会重用到它，并将说明 Timer1 外部模块作为中断源的用途。下面首先来编写中断服务子程序函数：

```
// 1. Timer1 interrupt service routine
void _ISR _T1Interrupt( void)
{

    // insert your code here
    // ...

    // remember to clear the interrupt flag before exit
    _T1IF = 0;

} //T1Interrupt
```

宏_ISR 的使用和前面一样，记住要将函数类型和参数都定义为 void。此外，还要在退出函数前对中断标志位（_T1IF）清零，这点尤其重要。通常，应用程序代码是很简练的。任何中断服务子程序的目标都是为了对某一事件作出快速高效的响应动作。作为惯例，如果读者发现自己的代码超过了一页（或者打算调用其他函数），就应该停下来重新考虑应用程序的目标和结构。冗长的计算部分应该放在主函数，特别是主循环里，而不是放在中断服务子程中。

下面加上主函数的一些代码来完成模板：

```
main()
{
    // 2. initializations
    _T1IP   = 4;           // set Timer1 priority, (4 is the default value)
    TMR1    = 0;           // clear the timer
    PR1     = period-1;    // set the period register

    // 2.1 configure Timer1 module clock source and sync setting
    T1CON   = 0x8000;      // check T1CON register options

    // 2.2 init the Timer1 Interrupt control bits
    _T1IF = 0;             // clear the interrupt flag, before
    _T1IE = 1;             // enable the T1 interrupt source

    // 2.3 init the processor priority level
    _IP = 0;               // 0 is the default value

    // 3. the main loop
    while( 1)
    {

            // your main code here...

    }       // main loop

} // main
```

在程序段 2，给 Timer1 中断源分配优先级，不过这不是必要的，因为每个中断源在通电后都有默认值为 4 的优先级。同时对定时器清零，并给周期寄存器分配初始值。

在程序段 2.1，完成了定时器模块的配置，并使用选择的设置启动定时器。

在程序段 2.2，在使能中断源之前，对中断标志位清零。

用于定时器模块的中断触发事件被定义为定时器值达到周期定时器中的预设值。中断发生后，中断标志位将置 1，定时器将开始新一轮的计数。如果中断使能位也为 1，并且它的优先级高于处理器当前的优先级（_IP），那么中断服务子程序马上就会被调用。

在程序段 2.3，初始化处理器的优先级。再强调一次，这不是必要的，因为在通电后处理器的优先级默认值为 0。

在程序段 3，插入主循环代码。如果一切按计划进行，主循环将会一直执行，中断服务子程序也会周期性地被调用。

5.3.4 Timer1 应用实例

只要多加几行代码，上面的模板就能转化成一个更实用的例子，其中 Timer1 用于保持实时

时钟产生 0.1 s、1 s 和 1 min 的间隔时间。作为简单的可视化反馈，将使用 PORTA 的低 8 位作为秒的二进制显示。下面即是需要增加的代码。

- 在程序段 1 之前，要另外定义一些整型变量，用作秒和分的计数器：

```
int dSec = 0;
int Sec = 0;
int Min = 0;
```

- 在程序段 1.2，使用中断服务子程序对计数器作增量计算：

```
dSec++;
```

此处，还需要增加一些额外的代码用于秒和分的进位处理。

- 在程序段 2，设置 Timer1 的周期寄存器的值以在两次中断之间获得 0.1 s 的间隔时间(假设使用 32 MHz 的时钟)。

```
PR1 = 25000-1; // 25,000 * 64 * 1 cycle (62.5ns) = 0.1 s
```

- 设置 PORTA 的 lsb 为输出：

```
TRISA = 0xff00;
```

- 在程序段 2.1，设置 Timer1 的预分频器值为 1:64，以获得期望的周期。

```
T1CON = 0x8020;
```

- 在程序段 3，给主循环中加入代码，以便使用毫秒计数器的当前值不断地刷新 PORTA (lsb) 的内容。

```
PORTA = Sec;
```

至此，新的项目已经构建好，如下所示：

```
#include <p24fj128ga010.h>

int dSec = 0;
int Sec = 0;
int Min = 0;

// 1. Timer1 interrupt service routine
void _ISR _T1Interrupt( void)
{
    // 1.1 your code here
    dSec++;                  // increment the tens of a second counter
    if ( dSec > 9)                  // 10 tens in a second
    {
            dSec = 0;
            Sec++;                  // increment the minute counter

            if ( Sec > 59)// 60 seconds make a minute
            {
                    Sec = 0;

    // 1.2 increment the minute counter
```

```
                Min++;

                if ( Min > 59)// 59 minutes in an hour
                    Min = 0;
            } // minutes
        } // seconds

        // 1.3 clear the interrupt flag
        _T1IF = 0;

    } //T1Interrupt

    main()
    {
        // 2. init Timer 1, T1ON, 1:1 prescaler, internal clock source
        _T1IP = 4;      // this is the default value anyway
        TMR1 = 0;               // clear the timer
        PR1 = 25000-1;// set the period register
        TRISA = 0xff00;         // set PORTA lsb as output

        // 2.1 configure Timer1 module
        T1CON = 0x8020;         // enabled, prescaler 1:64, internal clock

        // 2.2 init the Timer 1 Interrupt, clear the flag, enable the source
        _T1IF = 0;
        _T1IE = 1;

        // 2.3 init the processor priority level
        _IP = 0;                // this is the default value anyway

        // 3. main loop
        while( 1)
        {
            // your main code here
            PORTA = Sec;

        } // main loop

    } // main
```

5.3.5 Timer1 中断的测试

(1) 打开监视（Watch）窗口（把它放在合适的位置）。

(2) 加入以下的变量：

- dSec，从 Symbol（符号）下拉框中选择，然后按 Add（添加）按钮；
- TMR1，从 SFR 下拉框中选择，然后按 Add（添加）按钮；
- SR，从 SFR 下拉框中选择，然后按 Add（添加）按钮。

(3) 打开仿真器 Stopwatch（秒表）窗口（“Debugger→StopWatch”）。

(4) 在 1.1 部分之后的中断服务子程序的第一条指令处设置断点。

(5) 将光标置于该命令行，从右键菜单中选择“Set Breakpoint”（设置断点），或者直接双击。在这里设置断点，就能观察到中断是否被实际触发。

(6) 运行（"Debugger→Run"或者 F9）。仿真器很快就会停止，程序计数器光标（绿色箭头）停留在中断服务子程序的断点处。

于是，程序就停止在了中断服务子程序里面！也就是说，触发事件发生了，即 Timer1 计数到了 24 999（记住，Timer1 是从 0 开始计数的，因此已经数了 25 000 次）。再将 Timer1 的计数值乘以预分频器因子，即 25 000 × 64，也就是花费了 160 万个指令周期。

通过 StopWatch 窗口可以发现，实际执行的总指令周期数略大于 160 万。因为 StopWatch 中计算的周期数包含了程序的初始化部分。以 PIC24 的执行速度（每秒 1 600 万次或每周期 62.5 ns），完成以上的指令周期数只需要 0.1 s！

通过 Watch 窗口，可以看到当前处理器的优先级（IP）。由于是在优先级为 4 的中断服务子程序中，所以能够验证状态寄存器（SR）的第 3、4、5 位显示的就是该值。为方便起见，MPLAB IDE 在主窗口的底部有一个状态条，里面显示了状态寄存器的全部解码内容。

图 5-1 圈出了状态条中的 IP 值（IP4 表示中断优先级是 4）、SR 和秒表（用毫秒表示）的确切值。从当前位置单步执行（使用 StepOver 或 StepIn），就可以监视中断服务子程序中后续指令的执行。完成中断服务子程序后，可以看到优先级返回到了初始值——在状态条里发现 IP0 以及 SR 寄存器的第 5、6、7 位都被清零了。

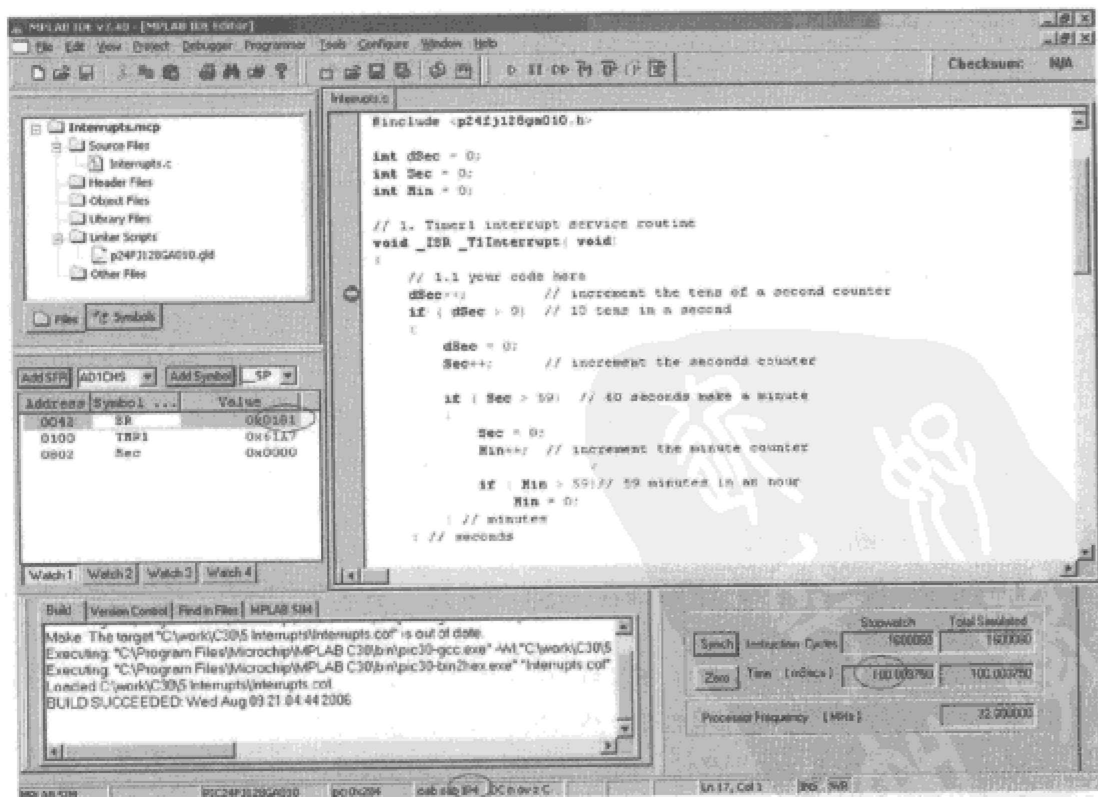


图 5-1　Timer1 中断后处理器状态的屏幕截图

(7) 再执行一次 Run 命令，将会发现程序计数器（用绿色箭头表示）再次停在了中断服务

子程序中。这次可以看到，160 万个指令周期已经增加到过去的计数器中。

(8) 在 Watch 窗口中加入 Sec 和 Min 变量。

(9) 执行多次的 Run 命令，经过 10 次后，可以发现秒计数器的值已经加 1。

为了测试分的增加，可以将当前的断点移动到下面几行的新位置——否则就要执行 600 次的 Run 命令了！

(10) 将新断点设置在 1.2 部分的 `Min++`语句上。

(11) 执行 Run 命令，可以发现秒计数器已被清零。

(12) 执行 StepOver 命令，可以发现分计数器的值已经加 1。

中断服务子程序一共执行了 600 次，间隔时间正好是 0.1 s。同时，主循环的代码已连续地执行了 9.6 亿个指令周期。诚然，该演示程序并没有完全利用到全部的指令周期——很多时间浪费在 PORTA 内容的更新上了。实际应用中，在保持准确的实时时钟计数的同时，还可以执行很多的工作。

5.3.6 二级振荡器

PIC24 的 Timer1 模块还有另一个特性（同以前所有的 8 位 PIC 微控制器一样），那就是用于维持实时时钟。实际上，可以使用一个低频振荡器（通常叫做二级振荡器）代替高频主时钟来为 Timer1 提供输入。由于它是针对低频操作（通常是和廉价的 32 768Hz 晶振一起使用）而设计的，所以它需要的电量很少。又由于它独立于主时钟电路，所以在主时钟不能工作或者处理器进入低功耗模式时，它都可以继续工作。事实上，二级振荡器是众多低功耗模式的必需组成部分。在一些情况下，可以使用它来代替主时钟，而另一些情况下，它只用作 Timer1 的输入或者是可选的外围部件。

要将前面的例子换成使用二级振荡器的情况，只需作很少的一些改动，例如以下改动。

- 将中断服务子程序改成只对秒和分计数(对于更低的时钟频率，不需要使用 0.1 s 计数)：

```
// 1. Timer1 interrupt service routine
void _ISR _T1Interrupt( void)
{
    // 1.1 clear the interrupt flag
    _T1IF = 0;

    // 1.2 your code here
    Sec++;          // increment the seconds counter

    if ( Sec > 59)// 60 seconds make a minute
    {
            Sec = 0;
            Min++; // increment the minute counter
            if ( Min > 59)// 59 minutes in an hour
                    Min = 0;
    } // minutes
} //T1Interrupt
```

- 在程序段 2，改变周期寄存器的值，每 32 768 个周期产生一次中断

```
PR1 = 32768-1;// set the period register
```

❑ 在程序段 2.1，改变 Timer1 的配置（不再需要预分频器）

```
T1CON = 0x8002;        // enabled, prescaler 1:1, use secondary oscillator
```

遗憾的是，读者不能马上使用仿真器测试这个新特性，因为二级振荡输入是不能自动模拟的。

在后面的章节中，将会介绍新的工具来生成仿真文件，以方便地把 32kHz 晶振连接到 PIC24 的 T1CK 和 SOSCI 引脚上。

5.3.7 实时时钟日历（RTCC）

基于前面两个例子，就可以开发具有完备功能的日历，包括日、周、月和年的计数。这些新增的代码行只会在每天、每周、每月或者每年执行一次，因此无论如何是不会降低整个程序的执行性能的。虽然开发这些代码是很有趣的，但是鉴于要经过几年的时间才能看到全部的执行细节，PIC24FJ128GA010 已经内置了一个完整的实时时钟日历模块，可以随时使用。真是方便啊！它不仅使用相同的低频二级振荡器，而且还配有响铃和鸣笛，以及能产生中断的内置报警功能。换而言之，当模块初始化完毕时，它就可以产生 RTCC 警报，等待中断发生，例如在每年预设的月、日、时、分、秒（如果将时间设定在 2 月 29 日，那就是每四年发生一次!）。

中断服务子程序可采用下面的形式：

```
// 1. RTCC interrupt service routine
void _ISR _RTCCInterrupt( void)
{
        // 1.1 clear the interrupt flag
        _RTCIF = 0;

        // 1.2 your code here, will be executed only once a year
        // that is once every 365 x 24 x 60 x 60 x 16,000,000 MCU cycles
// that is once every 504,576,000,000,000 MCU cycles

} // RTCCInterrupt
```

5.3.8 多个中断的管理

嵌入式控制应用通常要求服务多个中断源。例如，串行通信端口可能在 PWM 模块被激活并需要周期性的更新以控制模拟输出的同时，也需要周期性的监视。当多个输入被模数转换器采样，而且它们的采样值需要缓冲处理时，多个定时器模块可能同时被用来产生脉冲输出。对于有 118 个中断源的 PIC24 来说，几乎没有数量的限制。同时，如果忽略了某些规则，对于出现多个错误的情况，它似乎也是没有限制的，因为有同样精细的机制作保证。

以下是需要读者记住的一些规则。

(1) 保持简短。要保证中断服务子程序尽可能地最短/最快，在任何情况下它都不应该用来处理输入数据。要限制缓冲、传输和作标记之类的操作。

(2) 使用优先级来决定哪个事件先被处理，避免两个事件被同时触发。

(3) 要认真思考是否能承受嵌套中断调用带来的复杂度增大的情况出现。毕竟，如果中断服务子程序简短而高效，那么等待当前中断完成所引起的额外延时就会非常小。如果用户认为嵌套不是很必要，那么就要设置 NSTDIS 控制位来禁止嵌套：

```
_NSTDIS = 1; // disable interrupt nesting (default)
```

5.4 飞后小结

从本章可以知道，正是因为 C30 编译器内置的语言扩展和 PIC24 架构提供的强大的中断控制机制，中断服务子程序的编码才是如此地简单。对于嵌入式控制程序员来说，中断是一个非常方便有用的工具，尤其是在保持准确计时和资源受限的情况下能够管理多个任务。同时，它们也是麻烦的制造者。在 PIC24 参考手册和 MPLAB C30 用户指南中，读者可以参阅更多的有用信息。最后，本章介绍了更多关于 Timer1 和二级振荡器的内容，还有实时时钟日历（RTCC）模块的新特性。

5.5 给 C 语言专家的提示

中断向量表（IVT）是 PIC24 的 c0 代码的重要组成部分。实际上，在程序存储器的前 256 个地址里面有两个中断向量表的副本：一个用于正常程序的执行，另一个（或者是备用的 IVT）用于调试。这些表占据了前 5 章中所有例子的大部分 c0 码。从每个例子文件大小减去 256 个字（或者 768 字节），可以得到每个例子的净代码大小。

5.6 给汇编语言专家的提示

宏_ISRFAST 可以用来定义中断服务子程序函数，并进一步说明它会用到 PIC24 结构的一个新增的便利特性：一组 4 个的屏蔽寄存器。这允许处理器自动保存前 4 个工作寄存器（例如 w0~w3，最常使用的一组）和大部分 SR 寄存器的内容到特殊保留区域，而不需要使用栈，因此屏蔽寄存器能提供尽可能快速的中断响应。当然，由于只有一组这样的寄存器，所以它们每次只能满足一个中断的使用。虽然这并不限制整个程序只使用一个中断，但是对于所有中断都是相同优先级的应用程序不能仅仅使用_ISRFAST，或者当使用多个优先级时，要将_ISRFAST 预留给具有最高优先级的中断服务子程序。

5.7 给 PIC 微控制器专家的提示

注意，在 PIC24 结构中并没有一个可以用来禁止所有中断的控制位，不过有一个指令（DISI）可以在一定指令周期内禁止中断。如果有一部分代码要求所有中断临时被禁止，则可以使用下面的内联汇编命令：

```
__asm__ volatile("disi #0x3FFF"); // disable temporarily all interrupts

// your code here
// ...

DISICNT = 0; // re-enable all interrupts
```

5.8 提示与技巧

根据 PIC24 数据表，若要激活二级低功耗振荡器，用户需要将 OSCCON 寄存器的 SOSCEN 位置 1。不过，在录入上一个例子的代码并试图在真实目标板上执行前，要注意 OSCCON 寄存器是由锁定机制保护的，它里面包含着影响 MCU 选择主要的活跃振荡器及其速度的关键控制

位。为安全起见，用户需要先执行特殊的解锁程序，否则用户指令会被忽略。下面的这个例子就使用了内联汇编：

```
// OSCCON unlock sequence, setting SOSCEN
    asm volatile ("mov    #OSCCON,W1");
    asm volatile ("mov.b  #0x46, W2");
    asm volatile ("mov.b  #0x57, W3");
    asm volatile ("mov.b  #0x02, W0");    // SOSCEN =1
    asm volatile ("mov.b  W2, [W1]");
    asm volatile ("mov.b  W3, [W1]");
    asm volatile ("mov.b  W0, [W1]");
```

类似的组合锁定机制用来保护关键的 RTCC 寄存器 RCFGCAL。特殊位（RTCWREN）必须是 1 时，才能对寄存器进行改写，但是该位也需要先执行特殊的解锁程序。下面的例子再次用到了内联汇编代码：

```
// RCFGCAL unlock sequence, setting RTCWREN
asm volatile("disi      #5");
asm volatile("mov #0x55, w7");
asm volatile("mov w7,_NVMKEY");
asm volatile("mov #0xAA, w8");
asm volatile("mov w8,_NVMKEY");
asm volatile("bset      _RCFGCAL, #13");    // RTCWREN =1;
asm volatile("nop");
asm volatile("nop");
```

经过上面两步完成对 RTCC 的初始化后，日期和时间的设置就显得很简单了：

```
_RTCEN = 0;          // disable the module

// example set 12/01/2006 WED 12:01:30
_RTCPTR = 3;         // start the loading sequence
RTCVAL = 0x2006;     // YEAR
RTCVAL = 0x1100;     // MONTH-1/DAY-1
RTCVAL = 0x0312;     // WEEKDAY/HOURS
RTCVAL = 0x0130;     // MINUTES/SECONDS

// optional calibration
//_CAL = 0x00;

// enable and lock
_RTCEN = 1;          // enable the module
_RTCWREN = 0;        // lock settings
```

警报设置不需要特殊的解锁组合。下面的这个例子可以让读者记住作者的生日：

```
// disable alarm
_ALRMEN = 0;

// set the ALARM for a specific day of the year (my birthday)
_ALRMPTR = 2;        // start the sequence
ALRMVAL = 0x1124;    // MONTH-1/DAY-1
ALRMVAL = 0x0006;    // WEEKDAY/HOUR
ALRMVAL = 0x0000;    // MINUTES/SECONDS
```

```
// set the repeat counter
_ARPT = 0;            // once
_CHIME = 1;           // indefinitely

// set the alarm mask
_AMASK = 0b1001;      // once a year

_ALRMEN = 1;          // enable alarm
_RTCIF = 0;           // clear interrupt flag
_RTCIE = 1;           // enable interrupt
```

5.9 练习

编制中断服务子程序，完成下面的任务：

(1) 串行端口的软件仿真；

(2) 远程控制无线电接收机；

(3) NTSC 视频输出（提示：在后面的章节里，读者可以找到答案）。

5.10 推荐书目

- Curtis, K. E. (2006)
  Embedded Multitasking
  Newnes, Burlington, MA
  Keith 知道如何处理多任务，并且利用多任务构建简短而高效的嵌入式控制应用。
- Brown, G.(2003)
  Flying Carpet, The Soul of an Airplane
  Iowa State Press, Ames, IO
  作为总飞行员的 Greg，无论是将他的飞机作为生产工具时还是为家庭使用时，都有很多有趣的经历。

5.11 网上链接

- http://www.aopa.org
  这是飞机拥有者和飞行员协会的网站。读者可以随意浏览、阅读协会提供的许多杂志和享受协会提供的免费服务。相信读者会发现很多有趣的实用信息。

第 6 章　剖 析 引 擎

本章内容

- ▶存储器空间分配
- ▶程序空间可视化
- ▶存储器分配
- ▶查看 MAP 文件
- ▶指针
- ▶堆
- ▶MPLAB C30 存储器模型

学员正在努力获取的无论是驾驶执照还是飞行执照，迟早都必须要研究机罩下的引擎。学员无需知道引擎的每一部分是如何工作的，或者是怎样安装的——这将由机械工程师专业地处理。但是对引擎的大致了解可以帮助学员成为更优秀的驾驶员/飞行员。道理很简单，当了解了机器后，就可以更好地控制它，可以诊断一些小的故障，并且进行简单的维保。

使用编译器也没有什么不同。为获得最佳的性能，程序员早晚都应弄清机罩下引擎的结构。尽管在第 1 章中已经粗略地介绍过引擎的间隔室，但是本章将会有更深入的介绍。

6.1　飞行计划

本章将首先回顾字符串定义的基本知识，然后会介绍 MPLAB C30 编译器所使用的存储器分配技术。PIC24 的 RISC（精简指令集计算机）结构提出了一些有趣的挑战性问题并给出了创新性的解决方案。这里将会用到包括反汇编列表窗口、程序的存储器窗口和 MAP 文件等辅助手段来研究 MPLAB C30 编译器和连接器是如何联合生成最简洁而高效的程序代码的。

6.2　飞前备忘录

本章只用到软件工具，包括 MPLAB IDE 集成开发环境、MPLAB C30 编译器和 MPLAB SIM 仿真器。

使用“New Project Set-up”列表创建名为“Strings”（字符串）的新项目，同样地，创建名为“string.c”的源文件。

6.3　飞行

在 C 语言中，字符串被看作是简单的 ASCII 字符数组。假定组成字符串的每个字符都以数组的连续 8 位元素的形式顺序地存放在存储器中。在字符串的最末字符后，还附加有一个内容为零的字节（字符被表示为“\0”），作为结束标志。

注解 尽管可以很方便地直接使用标准C字符串操作库“string.h”。不过，要定义一个新的库，把字符串存放在第一个元素表示字符串长度的数组中，也是完全可能的——其实，这也是Pascal程序员们常用的方法。另外，如果用户开发的是一个“国际”项目（例如使用需要大字符集（如中文、日文或韩文）的语言进行通信的应用），那么就要考虑使用统一的字符编码标准（Unicode）来取代简单的ASCII码，这样可以给每个字符分配多个字节的空间。按照ANSI90标准，MPLAB C30编译器的“stdlib.h”库为多字节字符串的转换提供了最基本的支持。

首先来回顾一下单字符变量的定义：

```
char c;
```

同前面章节介绍的一样，上面的指令定义了一个8位整数(字符)，被默认为有符号数(−128~+127)。

用户也可以这样定义并使用数值给予初始化赋值：

```
char c = 0x41;
```

或者是使用ASCII码进行定义和初始化：

```
char c = 'a';
```

注意，要对ASCII字符常量使用单引号。上面两种定义的结果是一样的，对于C编译器来说，它们没有任何区别——字符就是数字。

下面来定义一个字符串，并初始化为8位整数（字符）的数组：

```
char s[5] = { 'H', 'E', 'L', 'L', 'O'};
```

上面的指令是使用数字数组标准格式来初始化的。不过，还可使用另一种更便捷的方法来初始化字符串：

```
char s[5] = "HELLO";
```

此外，还有一种更简单的方法，省去了用户计算字符串中字符个数的麻烦（也就避免了人为的错误），其格式如下：

```
char s[] = "HELLO";
```

MPLAB C30编译器自动添加结束字符（“0”），并确定字符串所需的字符数。正确的字符串长度，有利于后面的字符串操作。上面的指令实际上等价于下面的指令：

```
char s[6] = { 'H', 'E', 'L', 'L', 'O', '\0' };
```

给字符变量（8位整型）分配数值，执行算术运算与任何整型数的操作是相同的：

```
char c;// declare c as an 8-bit signed integer

c = 'a';    // assign to it the value corresponding to 'a' in the ASCII table
c ++;       // increment it... it will represent the ASCII character 'b' now
```

同样的操作可应用于任何字符数组的元素，不过就没有像上面初始化那样便捷的方法来对

整个字符串赋值：

```
char s[15];        // declare s as a string of 15 characters

s = "Hello!"; // Error! This does not work!
```

在源文件的开始处添加“string.h”文件，用户就可以用到很多有用的功能函数，包括以下几种。

- 将字符串的内容复制到另一个字符串：

```
strcpy( s, "HELLO");          // s : "HELLO"
```

- 添加（连接）两个字符串：

```
strcat( s, " WORLD");// s : "HELLO WORLD"
```

- 确定字符串的长度：

```
i = strlen( s);               // i : 11
```

- 还有更多的功能函数。

6.3.1　存储器空间分配

和数值初始化一样，当每次对字符串变量定义并以下面的形式初始化时：

```
char s[] = "Flying with the PIC24";
```

将会发生3件事情。

(1) MPLAB C30 连接器在 RAM（数据空间）为变量预留了连续的存储器地址：在上面的例子中是 22 个字节。这些空间属于 ndata（近）数据段。

(2) MPLAB C30 连接器将初始值保存在 22 字节长的表中（位于程序存储器中）。这些空间属于 init 代码段。

(3) MPLAB C30 编译器生成一个在 main 程序（前面章节已经提过的 c0 码的一部分）前被调用小程序，用于复制数据空间中的数值，从而完成变量初始化。

换而言之，字符串“Flying with the PIC24”实际占用的存储器空间是预想的两倍，一个复制是在 Flash 程序存储器里，另一个则是在 RAM 里。另外，还必须考虑到初始化代码和实际复制数据所花的时间。如果字符串不会在程序中再次改动，只是简单地传送到串口或者显示器，就不必浪费上面宝贵的资源。把字符串定义为“常量”，可以节省 RAM 存储器空间和初始化代码/时间：

```
const char s[] = "Flying with the PIC24";
```

这样，MPLAB C30 连接器只会分配程序存储器中的 const 代码段空间，字符串的访问可使用程序空间可视化窗口（Program Space Visibility window）——PIC24 结构的这一高级特性很快就会介绍。

在本章前面的例子里面，已介绍过将字符串隐性地定义成常量的情况：

```
strcpy( s, "HELLO");
```

字符串“HELLO”就被隐性地定义成 const char 类型，并只分配程序存储器中的 const 段空间，通过程序空间可视化窗口可以查看。

要注意，如果同一个常量字符串在程序中多次出现，MPLAB C30 编译器即使在所有优化功能都关闭的情况下，也会自动地保存一个副本在 const 段中，以优化存储器空间的使用。

6.3.2 程序空间可视化

PIC24 的结构和读者已经熟悉的大多数 16 位微控制器的结构可能会有些不同。它的设计目标是最高效率地使用哈佛结构模型，与之相对的是更常见的冯·诺伊曼结构模型。它们之间最大的区别在于，前者有两条完全独立的数据总线，一条用于访问程序存储器（Flash），而另一条用于访问数据存储器（RAM）。这一结构最明显的作用就是具有双倍的带宽，当一条指令在使用数据总线时，程序存储器总线就可以获取下一条指令的代码并开始译码。在传统的冯·诺伊曼结构中，这两种操作必须是交叉进行的，因此会带来性能上的损失。而哈佛结构的缺点是，访问程序存储器中存放的常量和数据时需要做些特殊的考虑。

PIC24 提供了两种读取程序存储器中数据的方法：使用特殊的表读取指令（tblrd）和使用叫作程序空间可视化或者 PSV 的第二种机制。PSV 是可从数据存储器总线访问的程序存储器内高达 32KB 大小的窗口。换而言之，PSV 是连接程序存储器总线和数据存储器总线的桥梁，如图 6-1 所示。

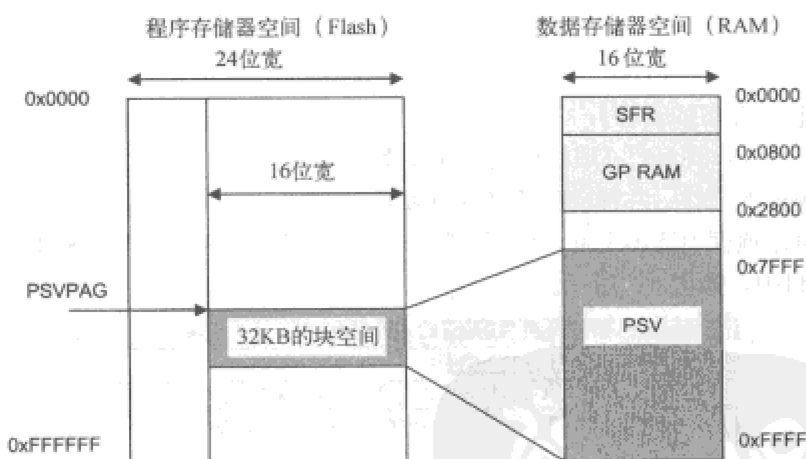


图 6-1 PIC24FJ128GA010 程序空间可视化窗口

注意，尽管 PIC24 使用 24 位宽的程序存储器总线，但是数据总线只有 16 位宽。这两条总线的不匹配使得 PSV“桥”很有意义。实际上，PSV 只把程序存储器的低 16 位连到数据存储器总线就可以。程序存储器的高 8 位对于 PSV 窗口来说是不可访问的。相反，当使用表访问指令时，程序存储器的所有位都是可访问的，不过就要注意操作 RAM 中数据（使用直接寻址）和操作程序存储器中数据（使用特殊表访问指令）的不同。

因此，PIC24 的程序员有两种选择，一种是更方便但存储器访问效率较低的方法，如 PSV，来实现两条总线间的数据传递；另一种是存储器访问效率更高、但透明度不高的表访问指令。

MPLAB C30 编译器的设计者考虑到了两者的折中并采用了双重机制，在不同的时刻用来解决不同的问题。

- PSV 用于处理常量数组（数字或字符串），因此对于常量和变量都是用同一类型的指针

（指向数据存储器总线）。

- 表访问机制用于执行变量的初始化（这受 c0 段的限制），以获得最大的代码紧凑度和高效率。

6.3.3　存储器分配

现在使用 MPLAB SIM 仿真器来开始研究存储器的分配，见如下代码段：

```
/*
** Strings
*/

#include <p24fj128ga010.h>
#include <string.h>

// 1. variable declarations
const char a[] = "Learn to fly with the PIC24";
char b[100] = "";

// 2. main program
main()
{
    strcpy( b, "MPLAB C30");     // assign new content to b
} //main
```

接下来，执行以下的步骤。

(1) 使用“Project Build”（项目构建）列表建立新项目。

(2) 添加 Watch 窗口（并移动到适当的位置）。

(3) 从符号选项框中选出变量“a”和“b”，并单击“Add Symbol”（添加符号），把它们加入到 Watch 窗口中，如图 6-2 所示。

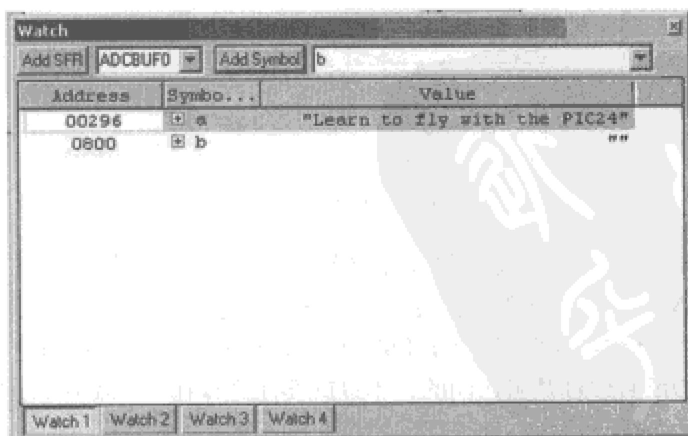


图 6-2　向 Watch 窗口加入数组

窗口中的“+”表示变量定义为数组，可以展开来给数组的每个独立元素赋值，如图 6-3 所示。

虽然数组的各项由 MPLAB 默认显示成 ASCII 字符形式，但是用户可以根据个人喜好改变

其显示方式。

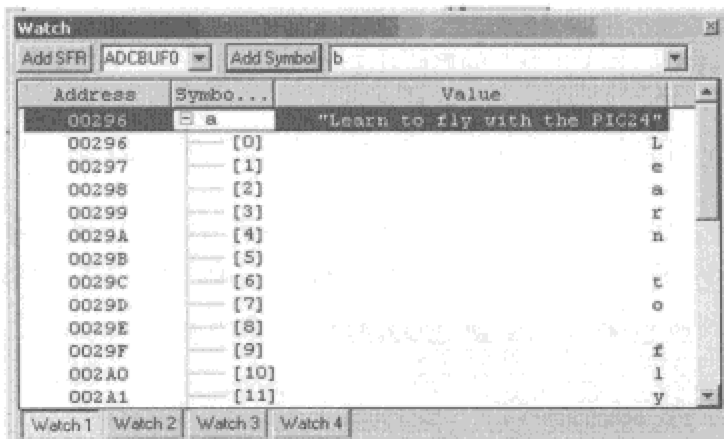


图 6-3　在 Watch 窗口中展开数组

(4) 用鼠标左键选中数组中的一个元素。

(5) 右键单击打开 Watch 窗口菜单。

(6) 选择“Properties”（属性）（菜单的最后一项）。

这样，就弹出 Watch 窗口的属性对话框，如图 6-4 所示。

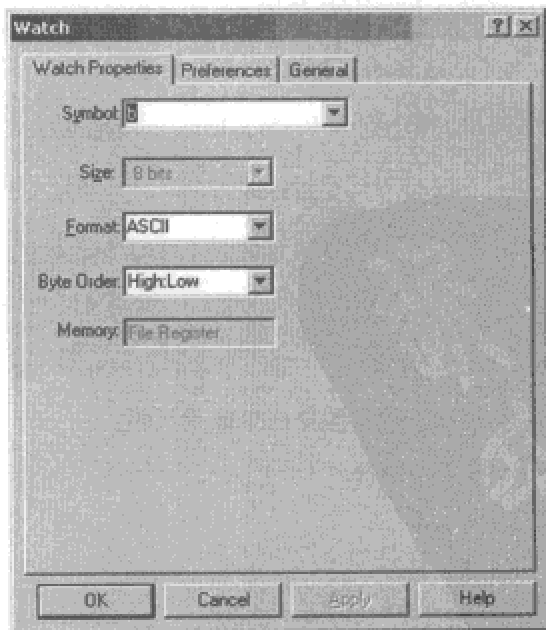


图 6-4　Watch 窗口属性对话框

在这个对话框中，用户可以改变所选定数组元素的显示格式，也可以查看“Memory”（存储器）项，获取所选变量的所在位置——数据段还是代码段。

如果打开常量字符串“a”的属性对话框，那么可以看到存储器一项显示的是“Program”（程序），这证实了在 PIC24 中常量字符串仅占用最少数量的 Flash 程序存储器空间，它可以通过 PSV 访问，而不需要占用 RAM。

相反，如果打开的是字符串“b”的属性对话框，那么会看到它位于文件寄存器或者其他的 RAM 存储器中。

继续往下探究，可注意到字符串“a”已完成初始化。在项目建立后，Watch 窗口马上就显示可以使用。

相反地，字符串“b”还是空的，尚未被初始化。只有当把光标放在主程序代码的第一行，运行“Run to Cursor”命令时，字符串“b”才会被赋予正确的值，如图 6-5 所示。

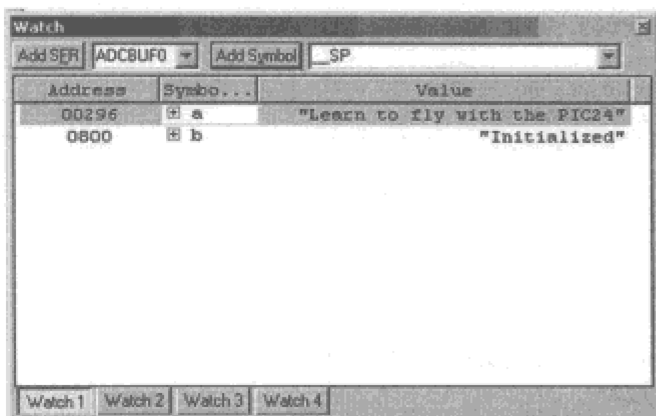


图 6-5　数组“b”初始化

正如前面所见，因为字符串“b”位于 RAM，所以必须先执行 c0 码段，才能对变量初始化以准备就绪。

提醒　Watch 窗口将所有的字符串靠右对齐，当一个字符串比较长（如本例子的“a”）而窗口比较窄时，用户可能就看不到其他较短的字符串。因此，如果有必要，可以移动 Watch 窗口并调整其窗口大小直至可以看到所有的项。

现在再次利用反汇编列表窗口，查看编译器生成的代码：

```
---  C:\work\C30\6 Strings\Strings.c  ------------------------------------------
1:                  /*
2:                  ** Strings
3:                  */
4:
5:                  #include <p24fj128ga010.h>
6:                  #include <string.h>
7:
8:                  // 1. variable declarations
9:
10:                 const char a[] = "Learn to fly with the PIC24";
11:                 char b[100] = "Initialized";
```

```
12:
13:                       // 2. main program
14:                       main()
15:                       {
 0028A   FA0000           lnk #0x0
16:                         strcpy( b, "MPLAB C30");     // assign new content to b
 0028C   282B21           mov.w #0x82b2,0x0002
 0028E   208000           mov.w #0x800,0x0000
 00290   07FFF7           rcall 0x000280
17:
18:                       } // main
 00292   FA8000           ulnk
 00294   060000           return

---  c:\pic30-build\build_20060131\src\standardc\sxl\strcpy.c  ----------------

 00280   780100           mov.w 0x0000,0x0004
 00282   784931           mov.b [0x0002++],[0x0004]
 00284   E00432           cp0.b [0x0004++]
 00286   3AFFFD           bra nz, 0x000282
 00288   060000           return
```

可以看到，在列表的底部，main()函数和strcpy()库函数已被完全反汇编。

注意 strcpy 子程序的代码多么简练，只有 5 条指令。读者还会注意到这是唯一的子程序。尽管“string.h”库有几十个函数，且“string.h”文件包括了它们的所有定义，但是连接器会智能地选择实际会使用到的那个函数。

不过，初始化代码 c0 是反汇编列表窗口不能显示的。正如前面章节介绍过的，用户必须通过程序存储器窗口才能观察到它（建议使用底部的图形化视图标签）。有好奇心且细心的读者可能会发现，字符串“b”的初始化使用的是读表（tblrd）指令来提取程序存储器（Flash）中的数据，然后将读得的数据存放到数据存储器（RAM）的指定位置。

6.3.4 查看 MAP 文件

在设计过程中，还可以使用另外一个工具来帮助读者理解字符串是如何被初始化和存储的，该工具就是“MAP file”(MAP 文件)。它是由 MPLAB C30 连接器产生的文本文件，使用 MPLAB 编辑器就可以打开，可以帮助用户理解和解决存储空间分配等问题。

要找到这个文件，请从主项目目录（里面包括有项目的全部源文件）中查找。选择“File→Open”，然后开始浏览直到找到项目目录。在默认状态下，MPLAB 编辑器列出的是所有的“.c”文件，不过用户可以将文件类型改变为“.map”文件，如图 6-6 所示。

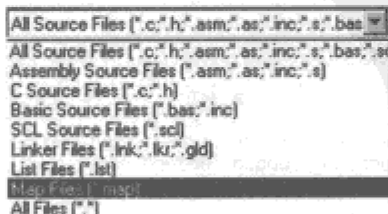


图 6-6 “.map”文件类型的选择

MAP 文件大多比较长，读者要学会查看其中一些重要的部分，这可以帮助读者发现很多有用的数据。例如，Program Memory Usage（程序存储器使用）的总数就可以在 MAP 文件的头几行中查到：

```
Program Memory Usage

section        address    length (PC units)    length (bytes) (dec)
-------        -------    -----------------    --------------------
.reset               0                  0x4           0x6  (6)
.ivt               0x4                 0xfc         0x17a  (378)
.aivt            0x104                 0xfc         0x17a  (378)
.text            0x200                 0x96          0xe1  (225)
.const           0x296                 0x26          0x39  (57)
.dinit           0x2bc                 0x4c          0x72  (114)
.isr             0x308                  0x2           0x3  (3)

        Total program memory used (bytes):          0x489  (1161) <1%
```

上面这个表是按特定的顺序(由.gld 连接器脚本文件决定)和位置列出了经过 MPLAB C30 连接器汇编的码段。

大多数码段的名字都是很直观的，而其他的沿用的是过去的名字。

- ❑ .reset 段放置的是复位向量。
- ❑ .ivt 是中断向量表，在第 5 章已经介绍过。
- ❑ .aivt 是备用中断向量表。
- ❑ .text 段放置经过 MPLAB C30 编译源文件后生成的代码（从最早的 C 编译器开始，这个名字就沿用至今）。
- ❑ .const 段放置需要通过 PSV 访问的常量（整型和字符串）。
- ❑ .dinit 段放置变量的初始化数据（c0 码使用）。
- ❑ .isr 放置中断服务子程序。

需要通过 PSV 窗口访问的常量字符串 “a” 和 “MPLAB C30”（隐含）常量字符串都保存在.const 段里。

读者可以通过查看程序存储器窗口的地址 0x296 来证实上面的结论。

要注意，对于 2×2 字符数组，PSV 是怎样仅仅使用程序存储器 24 位字中的 16 位的。

```
00290    ----    07FFF7  FA8000  060000  00654C   ........ ....Le..
00298    ----    007261  00206E  006F74  006620   ar..n .. to.. f..
002A0    ----    00796C  007720  007469  002068   ly.. w.. it..h ..
002A8    ----    006874  002065  004950  003243   th..e .. PI..C2..
002B0    ----    000034  00504D  00414C  002042   4...MP.. LA..B ..
002B8    ----    003343  000030  000800  000064   C3..0... ....d...
```

在.dinit 文件中，可以找到 “b” 变量初始化字符串。它是为表指令的访问作准备的，因此它使用了程序存储器中全部的 24 位字。要注意 3×3 字符数组的情况：

```
002C0    ----    000002  696E49  616974  7A696C   ....Ini. tia.liz.
002C8    ----    006465  000000  000000  000000   ed...... ........
002D0    ----    000000  000000  000000  000000   ........ ........
```

下面要考察的是 MAP 文件的另一部分，即 Data Memory Usage（数据存储器使用）(RAM)

的情况：

```
Data Memory Usage

section          address      alignment gaps    total length  (dec)
-------          -------      --------------    -------------------
.ndata             0x800                   0            0x64  (100)

               Total data memory used (bytes):          0x64  (100) 1%
```

在这个简单的例子中，只有一个码段：.ndata，里面只有一个变量“b”，而在 PIC24 的 RAM 中从地址 0x800 开始就为它保留了 100 个字节。

6.3.5 指针

指针是指用于间接访问（指向）其他变量或者自身某部分内容的变量。在 C 语言编程中，指针和字符串的紧密关系构成了任何数组数据类型的强大机制。实际上，正因为功能强大，所以它们也是程序员手上的最危险的工具之一，并是程序错误中非比例共享的根源所在。某些程序语言（例如 Java）已彻底禁止指针的使用，以保证程序语言的可靠性和可校验性。

MPLAB C30 编译器利用 PIC24 的 16 位结构来轻松地管理大量的数据存储器空间（在目前的版本中， RAM 空间的最大值是 32KB）。特别地，由于 PSV 窗口的缘故，导致 MPLAB C30 编译器将不会区分数据存储器目标的指针和程序存储器空间中 const 目标的指针。因此，这需要有一套标准的函数来处理两类存储器空间中的变量和/或存储块。

下面的经典程序例子将会比较使用指针和索引来顺序访问整型数组的区别：

```
int *pi;          // define a pointer to an integer
int    i;         // index/counter
int a[10];        // the array of integers

// 1. sequential access using array indexing
for( i=0; i<10; i++)
    a[ i] = i;

// 2. sequential access using a pointer
pi = a;
for( i=0; i<10; i++)
{
    *pi = i;
    pi++;
}
```

在程序段 1 中，执行简单的 for 循环，在每次循环中，使用变量 i 作为数组的索引。为了实现赋值操作，编译器需要提取变量 i 值，并将它乘以数组元素所占的字节数（2），然后将所得的偏移量加到数组“a”的初始地址上。

在程序段 2 中，将指针初始化为指向数组“a”的初始地址。在每次循环中，只需简单地使用指针（*）来执行赋值任务，然后将指针加 1。

从上述两种方法的对比中可以看到，使用指针可以在每次循环中节省乘法运算至少 1 次。如果在循环体中数组元素多次被使用，那么性能的改进效果将按比例增加。

C 语言中的指针语法是很简单的，虽然这可以使程序员编写出高效的代码，但同时它也为更多的程序错误大开城门。

读者至少应该熟悉一些最常用的简化代码。上面代码的其中一部分通常可简写成以下的形式：

```
// 2. sequential access to array using pointers
for( i=0, p=a; i<10; i++)
    *pi++ = i;
```

读者还要注意空指针（即没有任何目标的指针），它指向一个特殊的值 NULL，在"stddef.h"文件中有它的定义和实现说明。

6.3.6 堆

使用指针的其中一个好处就是，可以操作存储器中动态（在运行过程中）定义的对象。"堆"就是数据存储器中为这类操作预留的区域，而且在标准 C 库的"stdlib.h"中还提供了一组函数用以分配和释放存储块。其中最少应包括下面的基本函数：

```
void *malloc(size_t size);
```

它从堆中取出所需大小的存储块，然后返回一个指针。

```
void free(void *ptr);
```

将 ptr 指向的存储块释放给堆。

MPLAB C30 连接器把堆设置在所有项目全局变量和保留栈没有使用到的 RAM 空间。尽管连接器知道存储器中有着大量的未使用空间，并且在每个项目的 MAP 文件中都有罗列出，但是用户还是需要明确地指出连接器要为堆保留的空间大小。

使用"Project→BuildOptions→Project"菜单命令打开"Build Option"（构建选项）对话框，选择 MPALB Link30 标签，并定义堆的大小（字节数）。

通常应尽可能分配最大字节数的存储器空间，这样就可以使得 malloc() 函数最高效地利用可用的存储器空间。毕竟，如果不分配给堆，那么它们是不被使用的。

6.3.7 MPLAB C30 存储器模型

PIC24 结构允许在数据存储器的前 8KB 地址范围内使用一个非常高效（压缩）的指令编码来执行各种处理。这个地址范围称为存储器的"近"地址，对于 PIC24FJ128GA010 来说，它对应于 SFR 组（前 2KB）和紧接着的 6KB 的通用 RAM 空间。只有 RAM 最前面的 2KB 实际上是在"近"地址以外的。

要访问 8KB 以外的范围，就需要使用间接寻址方法（指针），如果处理得不好，访问效率是很低的。栈（和 C 函数用到的所有局部变量）和堆（用于动态存储器分配），自然是可以使用指针访问的，而且相应地最理想的方法就是把它们放置在 RAM 最前面的位置。这是连接器默认的处理方法。同时，连接器还会尽量把一个项目的所有全局变量放置在近地址空间，以获得最大的访问效率。如果一个变量不能放置在近地址空间，那么必须通过"手动"方式定义成"远"地址属性，于是编译器就会生成合适的访问代码。这类处理方式叫做"小数据存储器模型"

(与其相对的是“大数据存储器模型”)，每个变量都默认为“远”地址属性，除非被特别地定义成“近”地址属性。

其实，对于 PIC24FJ128GA010 微控制器，大多数情况下都是使用默认的小数据存储器模型，而仅在很少场合才需要识别变量的“远”地址属性。在第 12 章里，读者将看到这种情况，一个不能存放在近存储器空间的超大数组必须被定义成“远”地址属性。因此，不仅编译器能生成正确的寻址指令，而且连接器也能将该数组变量放置在 RAM 的前面位置，并给其他的变量赋予优先级，允许在近地址空间访问它们。

由于数组元素的访问是通过间接寻址进行的(具体来说是通过指针或者索引方式)，所以没有额外的性能和代码开销。

程序存储器空间也有类似的操作。实际上，对于每个编译模块，函数的调用都是通过压缩的寻址表来完成的，其最大的寻址范围是 32KB。程序存储器模块（大数据或者小数据）定义了编译器/连接器的默认操作，是在这 32KB 范围以内或者以外进行函数寻址的。

6.4 飞后小结

在 C 语言中，字符串被定义为简单的字符数组，不过 C 语言并没有明确的概念来区分不同的存储器区域（如 RAM 与 Flash)，也不需要特殊的机制来连接哈佛结构中的不同总线。程序员在使用 MPLAB C30 编译器时，需要对嵌入式控制应用中的多机制之间的平衡和用来最大限度地利用宝贵资源（特别是 RAM）的存储器分配策略有较为基本的理解。

6.5 给 C 语言专家的提示

const 属性在 C 语言中是经常用到的，配合大多数变量类型的使用，可以辅助编译器找出常见的参数使用错误。当一个参数以 const 形式传递给函数或者一个变量被定义为 const 时，编译器能够标记出对变量所作的改动。正如前面介绍的，MPLAB C30 使用 PSV 以非常自然的方式扩展了这种语义，使得代码的实现具有更高的效率。

6.6 给汇编语言专家的提示

函数库“string.h”包含了很多有用的块操作函数，通过指针就可以操作任何类型的数组而不单是字符串，如 memcpy()、memcmp()、memset()和 memove()。

函数库“ctype.h”包含有替换函数，用来根据 ASCII 表的位置来识别单个字符、区分大小写以及进行大小写的相互转换。

6.7 给 PIC 微控制器专家的提示

由于 PIC24 程序存储器的实现通常采用 Flash 技术，在代码执行期间，只需要单电压就可实现编程，因此设计出引导程序（boot-loader）(即用于自动更新全部或者部分代码的应用程序）是可能的。另外，在一些基本的限制条件下，还可以将 Flash 程序存储器的一个部分空间用作非易失性存储区域。Flash 程序存储器的写入，需要使用表访问方法，并且要特别地小心。PSV 窗口是一个可读设备，正如前面所见，它只能访问程序存储器地址 24 位中的 16 位。

6.8 提示与技巧

一旦读者掌握了结束字符 0 的有效使用，那么 C 语言中的字符串操作将是非常有趣的。例如，执行下面的 mycpy() 函数：

```
void mycpy( char *dest, char * src)
{
    while( *dest++ = *src++);
}
```

上面的代码片段是很危险的，因为它没有限制要复制的字符个数，也没有检查 dest 指针指向的缓冲器是否足够大，那么读者可以想象若 src 字符串缺失结束字符将会发生什么情况。该代码片段很容易在所分配的变量空间以外的位置继续运行，从而破坏数据 RAM 的整个内容（包括所有的 SFR）。

至少，读者应该在使用之前检查传递至函数的指针是否已被正确地初始化。将指针与 NULL 值（已定义在“stdlib.h”和/或“stddef.h”）进行比较，就可以捕捉到可能的错误。

对复制的字节数应设定限制。显然，读者应该知道程序中字符串/数组的长度。如果确实不知道，那么可以使用 sizeof()。函数 mycpy() 的更好实现如下：

```
void mycpy( char *dest, char *src, int max)
{
    if ((dest != NULL) && ( src != NULL))
        while (( max-- > 0) && ( *src))
            *dest++ = *src++;
}
```

6.9 练习

开发新的字符串操作函数，执行下列操作。

(1) 依次找出字符串数组中的每个字符串。

(2) 实现二进制的搜索。

(3) 开发一个简单的散列表管理函数库。

6.10 推荐书目

❑ Wirth, N. (1976)

Algorithms + Data Structures = Programs

Prentice-Hall, Englewood Cliffs, NJ

Wirth（Pascal 程序语言之父）以无可比拟的通俗语言带领读者实现从最简单的编程到设计自己的编译器的飞跃。

6.11 网上链接

❑ http://en.wikipediaorg/wiki/Pointers#Support_in_various_programming_languages

它将带领读者学习到更多关于指针的知识，观察在不同的编程语言中是如何管理指针的。

第二部分

单　　飞

恭喜！读者已经完成了前一阶段的课程，也具备了不再坐在教练身边而开始独自飞行所必需的勇气，可以单飞了。因此，下面的课程就更要看读者自己的表现了。

本书的第二部分将继续介绍 PIC24 与外部世界连接的基本外设。由于给出的示例程序有些复杂，建议读者准备好真实的演示板，以实现真正的仿真。本书推荐使用标准的 Microchip Explorer16 演示板，不过也可以使用能提供相似特性或者仿真原型的第三方工具。



第 7 章　通　　信

本章内容

- ▶同步串行接口
- ▶异步串行接口
- ▶并行接口
- ▶使用 SPI 模块的同步通信
- ▶读状态寄存器指令的测试
- ▶写 EEPROM
- ▶读存储器内容
- ▶永久性存储库
- ▶新 NVM 库的测试

一些大的航空公司有时会开辟额外的频道——“驾驶舱频道”。有了它，人们就可以通过无线电收听飞行员和交通控制员之间的真实对话。如果是初次听到，不可能觉得里面的对话有什么意义。里面的内容听起来就像是一连串随机的数字和难以识别的代号。但是，继续听下去，并且慢慢熟悉飞行的术语，对话内容就会逐渐明朗起来。飞行员和控制员都需要遵循精确的协议，即选择合适的无线电频率作为传输媒介，学习全套的通信语言并与不同飞机进行通信。

在嵌入式控制中，通信也就是要理解协议和物理媒介的使用特性。在嵌入式控制的编程中，学会选择正确的通信接口及懂得如何使用它们都是非常的重要。

7.1　飞行计划

本章将会介绍 PIC24 系列中所有的通用设备都会用到的通信接口。特别地，还会深入研究异步串行通信接口 UART1 和 UART2，以及同步串行接口 SPI1 和 SPI2，并比较它们在不同的嵌入式控制应用中的优缺点。

7.2　飞前备忘录

除了 MPLAB IDE、MPLAB C30 编译器和 MPLAB SIM 仿真器这些常用的软件工具外，本章还需要使用 Explorer16 演示板和 MPLAB ICD2 电路内调试器。

使用“New Project Set-up”列表生成名为“SPI”的新项目和名为“spi2.c”的新源文件。

7.3　飞行

PIC24FJ128GA010 提供了 7 个通信外设，用于支持嵌入式控制的所有应用。其中 6 个是“串行”通信外设，每次只发送和接收一个位信息，它们是：

- ❑ 2 个通用异步接收器和发送器；
- ❑ 2 个 SPI 同步串行接口；

- 2 个 I^2C 同步串行接口。

同步接口（如 SPI 和 I^2C）和异步接口（如 UART）的主要区别是从发送端到接收端的时序信息传递方式。同步通信接口需要物理链路（一条电线）来传输时钟信号，为两个设备提供同步信息。提供时钟信号的设备通常称为“Master”（主）设备，而与主设备保持同步的设备通常被称为“Slave”（从）设备。

7.3.1 同步串行接口

例如，I^2C 接口使用两条电线（因此使用微控制器的两个引脚），一条用于时钟（被称作 SCL），而另一条用于数据（被称作 SDA），如图 7-1 所示。

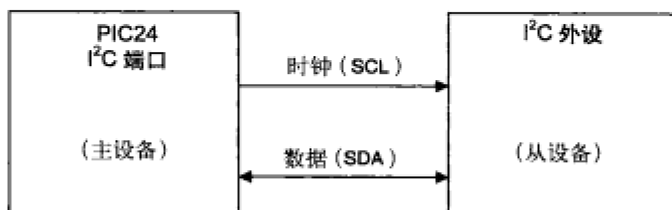


图 7-1 I^2C 接口模块图

代替的是，SPI 接口将数据线分成了两条，一条用于输入（SDI），而另一条用于输出（SDO），此外，还需要使用一条额外的电线来保证数据在两个方向上同时快速地传输，如图 7-2 所示。

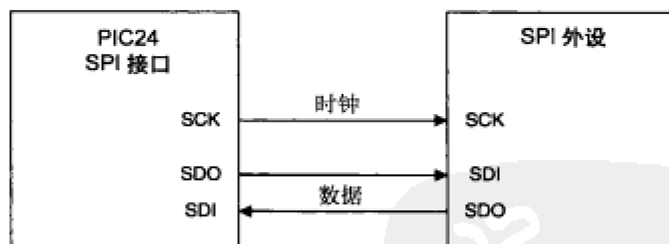


图 7-2 SPI 接口模块图

为了在同一串行接口上挂接多个设备（总线配置），I^2C 接口需要在数据开始传输之前通过数据线发送 10 位的地址。虽然降低了通信的速度，但是（理论上）允许两条线路（SCL 和 SDA）支持 1 000 个外部设备。同时，I^2C 接口也允许将多个设备作为“主”设备，使用仲裁协议实现总线的共享。

另一方面，SPI 接口需要额外的物理线路，以将“从设备选择（SS）”引脚连接到各个设备。实际上，这意味着在使用 SPI 总线（如图 7-3 所示）时，随着所连接设备的增加，PIC24 需要的 I/O 引脚数量也会成比例地增加。

在多个主设备之间共享 SPI 总线在理论上是可行的，但是在实际中很少这样应用。SPI 接口的主要优势是结构简单，并且速度快，甚至比最快的 I^2C 总线还要快一个数量级（这里没有考虑通信协议的解释程序）。

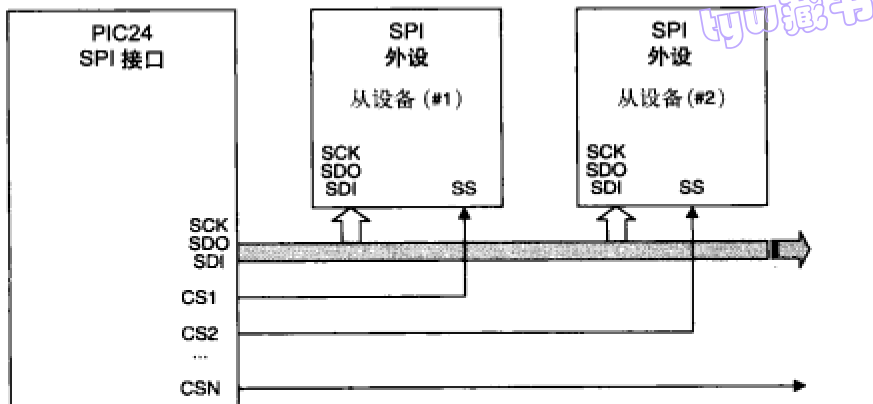


图 7-3　SPI 总线模块图

7.3.2　异步串行接口

异步通信接口是没有时钟线的，通常是使用两条数据线路：TX 和 RX，它们分别用于输出和输入（另外还有两条可选的线路，用于硬件握手信号），如图 7-4 所示。发送端和接收端之间的同步由数据流中的时序信息提供。将开始位和终止位添加到数据中，严谨的格式（使用特定的波特率）保证了数据传输的可靠性。

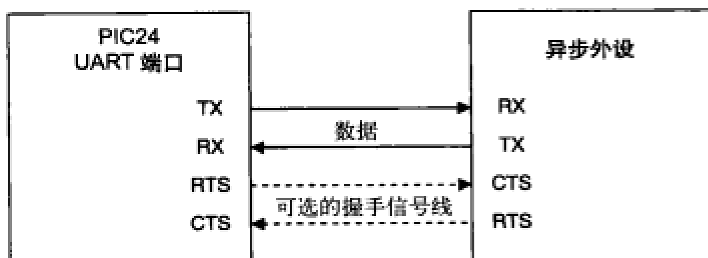


图 7-4　异步串行接口模块图

为提高抗噪特性，很多异步串行接口标准规定了收发器的使用，这样可以将传输距离延伸到几千英尺以外。

每一种串行通信接口都有自己的优势和劣势。表 7-1 总结列举了每种接口最重要的特性及其最常见的应用场合。

表 7-1　同步串行通信和异步串行通信外设的比较

	同步通信		异步通信
外设	SPI	I^2C	UART
最大位传输速度	10Mb/s	1Mb/s	500Kb/s
最大总线宽度	由引脚数决定	128 个设备	点到点（RS232） 256 个设备（RS485）
引脚数	3+n×CS	2	2
优势	简单、低耗、高速	引脚少，允许有多个主设备	距离长，改善了抗噪性能（需要使用发收器）

（续）

	同步通信		异步通信
劣势	单一主设备、距离短	速度最慢、距离短	需要精确的时钟频率
典型应用	在同一块 PCB 上直接连接到 ASIC 和其他外设	在同一块 PCB 上总线和外设相	可以连接终端、个人电脑和其他数据采集系统
应用实例	串行 EEPROMS(25CXXX 系列)，MCP320X A/D 转换器，ENC28J60 以太网控制器，MCP251X CAN 总线控制器……	串行 EEPROMS（24CXXX 系列），MCP98XX 温度传感器，MCP322X A/D 转换器……	RS232、RS422、RS485、LI 总线，MCP2550 IrDA 接口……

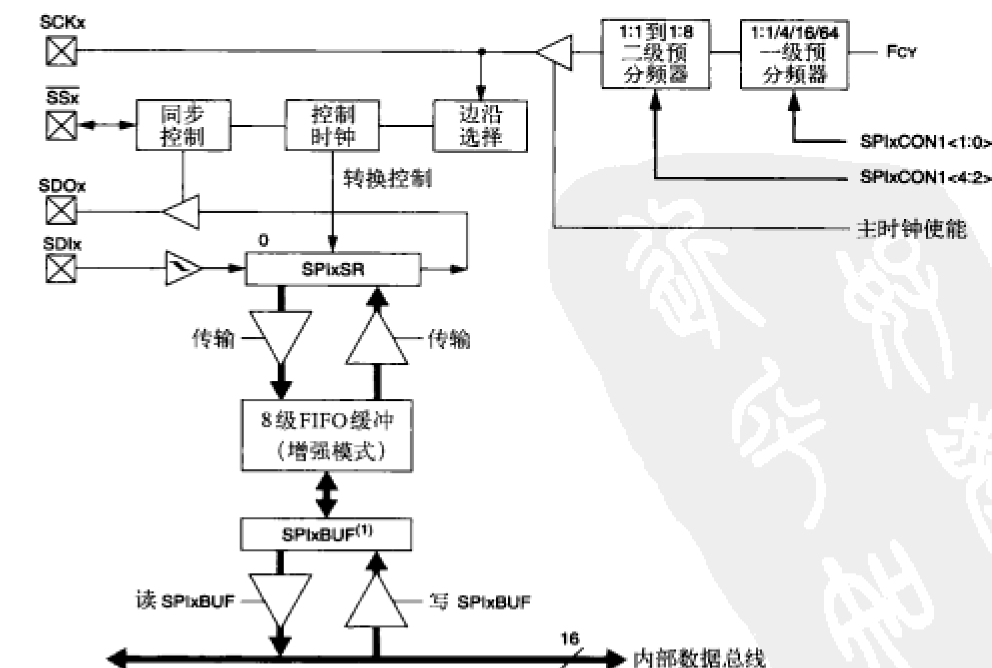
7.3.3 并行接口

并行主端口（PMP）完善了 PIC24 的基本通信接口。PMP 可以同时传输最多 8 位的信息，并且提供多条地址线，因此可以直接连接到大多数的 LCD 显示模块（含集成控制器的字母数字和图形模块），以及小型 Flash 存储卡（或者 CF-I/O 设备）、打印机端口和市场上绝大部分以“-CS、-RD、-WR”为标识的其他 8 位并行设备。

本章稍后将专门介绍其中一个同步串行接口：SPI。而在后面的章节中，将会再分别介绍异步串行接口和 PMP。

7.3.4 使用 SPI 模块进行同步通信

尽管 PIC24 的实现具有相当丰富的选择和有趣的特性，不过 SPI 接口应该是最简单的接口，其模块示意图如图 7-5 所示。



（1）在标准模式下，数据是直接在 SPIxSR 和 SPIxBUF 之间传送的。

图 7-5　SPI 模块示意图

SPI 接口主要由 8 位移位寄存器组成：所有位同时从 SDI 线移入（MSB 先移入），然后从 SDO 线移出，引脚 SCK 的时钟作为同步信号。

如果设备被设置为总线主控设备，在内部生成时钟（为追求最大限度的柔性设计，需使用两个串联的预分频器，从外部时钟提取），并且在 SCK 引脚输出。如果设备是总线从控设备，那么时钟将从 SCK 引脚提取。

在本章将要用到的所有其他外设，都有一个共同的特性，即受特殊功能寄存器 SPIxCON1（如图 7-6 所示）的控制以及 SPIxCON2 的高级选项的控制。

高字节：

U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	DISSCK	DISSDO	MODE16	SMP	CKE
位15							位8

低字节：

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SSEN	CKP	MSTEN	SPRE2	SPRE1	SPRE0	PPRE1	PPRE0
位7							位0

图 7-6 SPIxCON1 控制寄存器

为了说明 SPI 的基本功能，这里需要用到 Explorer16 演示板，其中 PIC24 的 SPI2 模块是连接到一个 25LC256EEPROM 设备的，通常是指串行 EEPROM（即 SEE 或者有时称作 E^2——读作“E 的平方”）。该设备小巧便宜，可提供 256Kb（或者 32KB）的非易失性高持久的存储器。

要实现 SPI2 模块和串行存储设备之间的通信，首先需要精心设置外部模块的配置。

按照设备数据表的介绍，SEE 对应一个 8 位（MOD16=0）命令，使用以下的设置并通过 SPI 接口提供工作电源。

- ❑ 时钟信号 IDLE 电平为低，时钟信号 ACTIVE 为高（CKP=0）。
- ❑ 在从 ACTIVE 到 IDLE 时，串行输出改变（CKE=1）。

PIC24 作为总线主控制设备（MSTEN=1），由内部时钟经过预分频器后产生时钟信号 SCK（在这里，使用的是默认预分频器值 1：64 和 1：8，得到总的预分频值为 1：512）。

将选定的配置值定义为常数，然后分配给 SPI2CON1 寄存器：

```
#define SPI_MASTER  0x0120      // select 8-bit master mode, CKE=1, CKP=0
```

为了使能外围设备，需要访问 SPI2STAT 寄存器，同其他大多数 PIC24 外设一样，它的第 15 位是主使能控制位，而其他位的常量则被设置为可读：

```
#define SPI_ENABLE 0x8000       // enable SPI port, clear status
```

将 PORTD 的引脚 12 连接到存储器的片选引脚（CS），低电平有效，因此若在程序中再添加两条定义，则程序更易读：

```
#define CSEE        _RD12       // select line for Serial EEPROM
#define TCSEE       _TRISD12    // tris control for CSEE pin
```

现在可以开始编写演示程序的外设初始化代码：

```
// 1. init the PIC24 SPI peripheral
TCSEE = 0;              // make SSEE pin output
CSEE = 1;               // de-select the Serial EEPROM (low power standby)
SPI2CON1 = SPI_MASTER;  // select mode
SPI2STAT = SPI_ENABLE;  // enable the peripheral
```

下面为实现串行 EEPROM 设备的数据收发编写简单的函数：

```
// send one byte of data and receive one back at the same time
int writeSPI2( int data)
{
    SPI2BUF = data;                  // write to buffer for TX
    while( !SPI2STATbits.SPIRBF);    // wait for transfer to complete
    return SPI2BUF;                  // read the received value
}//writeSPI2
```

函数 writeSPI2 是一个真正的双向传输函数。它可以立即写入一个字符到发送缓冲器，然后进入等待循环，直到接收标志位显示发送完毕。同样也可以从设备接收数据，接收到的数据作为函数值返回。

然而，在与存储设备通信时，会出现这样的情况：当命令发送给存储器后，不会立即得到响应，当从存储设备读取数据时，不再需要 PIC24 发送数据命令。在第一种情况（写命令）里，函数的返回值可以直接忽略。在第二种情况（读命令）里，当数据从设备传输时，会有一个空值传送给存储器。

25LC256 数据表准确地描述了用于存储设备读写的全部 7 条命令。使用如下的常数表格有助于对这些命令编程：

```
// 25LC256 Serial EEPROM commands
#define SEE_WRSR   1          // write status register
#define SEE_WRITE  2          // write command
#define SEE_READ   3          // read command
#define SEE_WDI    4          // write disable
#define SEE_STAT   5          // read status register
#define SEE_WEN    6          // write enable
```

下面，通过一个短小的测试程序，来验证同设备的通信是否已正确建立。例如，使用读状态寄存器命令，就可以访问存储器设备，并验证 SPI 外设是否配置合理。

7.3.5 测试读状态寄存器命令

读状态寄存器命令的完整时序图如图 7-7 所示。在发送适当的命令（SEE_STAT）后，还需要另外调用使用空值数据的 writeSPI2() 函数，以捕捉存储设备的响应。

向 SEE 发送任何命令，至少需要经过以下的步骤。

- 激活存储器，将 CS 引脚置为低电平。
- 移出 8 位指令。
- 根据不同的指令，这里需要额外的一个或多个步骤。
- 关闭存储器（将 CS 引脚置高电平），完成命令，存储器恢复到低功耗的等待状态。

实际上，下面的代码可用来实现完整的读状态寄存器操作：

```
// Check the Serial EEPROM status
CSEE = 0;                    // select the Serial EEPROM
writeSPI2( SEE_STAT);        // send a READ STATUS COMMAND, ignore immediate data
i = writeSPI2( 0);           // send dummy, read data
CSEE = 1;                    // deselect to complete the command
```

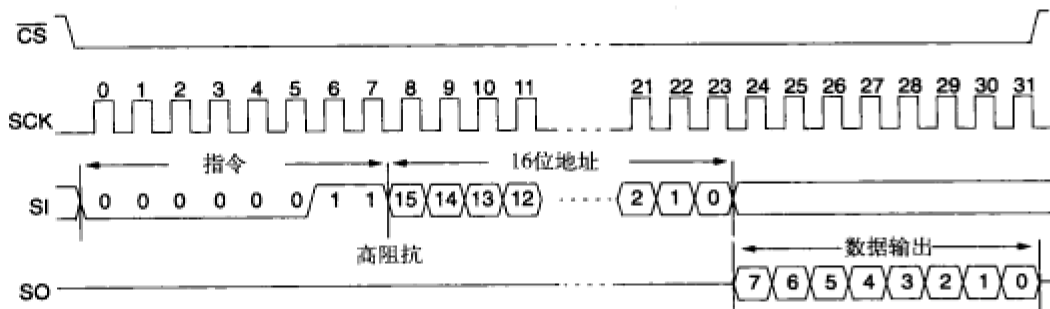


图 7-7　读状态寄存器命令的完整时序图

完整的项目程序如下所示：

```
/*
** SPI2 demo
*/
#include <p24fj128ga010.h>

// I/O definitions
#define CSEE    _RD12           // select line for Serial EEPROM
#define TCSEE   _TRISD12        // tris control for CSEE pin

// peripheral configurations
#define SPI_MASTER  0x0120          // select 8-bit master mode, CKE=1, CKP=0
#define SPI_ENABLE     0x8000       // enable SPI port, clear status

// 25LC256 Serial EEPROM commands
#define SEE_WRSR        1           // write status register
#define SEE_WRITE       2           // write command
#define SEE_READ        3           // read command
#define SEE_WDI         4           // write disable
#define SEE_STAT        5           // read status register
#define SEE_WEN         6           // write enable


// send one byte of data and receive one back at the same time
int writeSPI2( int data)
{
    SPI2BUF = data;                         // write to buffer for TX
    while( !SPI2STATbits.SPIRBF);           // wait for transfer to complete
    return SPI2BUF;                         // read the received value
}//writeSPI2

main()
{
        int i;
```

```
        // 1. init the SPI peripheral
        TCSEE = 0;                          // make SSEE pin output
        CSEE = 1;                           // de-select the Serial EEPROM
        SPI2CON1 = SPI_MASTER;              // select mode
        SPI2STAT = SPI_ENABLE;              // enable the peripheral

    // 2. Check the Serial EEPROM status
        CSEE = 0;                           // select the Serial EEPROM
        writeSPI2( SEE_STAT);               // send a READ STATUS COMMAND
        i = writeSPI2( 0);                  // send dummy, read data
        CSEE = 1;                           // terminate command <-set brkpt here

} // main
```

按照“MPLAB ICD2 Debugger Set-up（调试器设置）”列表，开启电路内调试器，进行项目设置。然后根据“Project Build”（项目建立）列表对演示程序进行编译和调试。

(1) 将 ICD2 连接到 Explorer16 演示板，选择“Debugger（调试）→Program（编程）”，对 PIC24 编程。在默认情况下，MPLAB 会选择所需的最小存储器范围来将项目代码传送到设备中，因此编程时间可达到最小化。只需要几秒钟的时间，PIC24 就可以完成编程和校验，进入准备执行状态。

(2) 添加 Watch window（监视窗口）到项目。

(3) 在符号选择框中选择“i”，然后单击“Add Symbol”（加入符号）按钮。

(4) 将光标置于主循环代码的最后一行，设置断点（双击）。然后使用“Debug→Run”命令开始运行程序。

(5) 当程序执行完毕，25LC256 存储状态寄存器的内容应该已经传送给变量“i”，这在 Watch window（监视窗口）就能看到。

令人失望的是，25LC256 存储器的状态默认值（通电后）是 0x00，因为 BP1 和 BP0 的关闭状态表明没有模块保护功能，禁止写锁存器 WEL 无效，在线写入 WIP 标志位无效。25LC256 串行 EEPROM 状态寄存器如表 7-2 所示。

上面测试程序的结果不是很理想。因此，为了增加测试的趣味，在查询状态寄存器前，先设置写锁存器——此时将会看到第 1 位已被置位。

表 7-2　25LC256 串行 EEPROM 状态寄存器

7	6	5	4	3	2	1	0
W/R	—	—	—	W/R	W/R	R	R
WPEN	x	x	x	BP1	BP0	WEL	WIP

W/R=读/写，R=只读

要对写锁存器置 1，需要在程序段 2 之前插入以下的代码，并立即重新编号为 2.2。

```
// 2.1 send a Write Enable command
CSEE = 0;                 // select the Serial EEPROM
writeSPI2( SEE_WEN);      // send command, ignore immediate data
CSEE = 1;                 // deselect to complete the command
```

(1) 重建项目。

(2) 重新编程设备。

(3) 在主程序的最后一行代码处设置断点。

(4) 运行（或者单击 Run to Cursor）。

如果一切顺利，就会在监视窗口中观察到变量“i”已变为红色，其值为 2。

这种巨大的成就感，只有在对 16 位嵌入式控制器进行编程时才能体会到！

注意，现在写锁存器标志位已经置 1，可以使用写命令来“改变”EEPROM 的内容了。用户可以每次写入一个字节，或者写入一个最大值为 64 字节的字符串，这些写入操作都可使用被称作页写入模式（Page Write）的序列/命令来实现。不过要注意数据表上关于该操作模式的地址限制。

7.3.6 写 EEPROM

在发送写命令后，必须在实际数据移出之前提供两个字节的地址（ADDR_MSB 和 ADD_LSB）。下面的代码给出了正确的写入序列范例：

```
// send a Write command
CSEE = 0;                // select the Serial EEPROM
writeSPI2( SEE_WRITE);   // send command, ignore immediate data
writeSPI2( ADDR_MSB);    // send MSB of memory address
writeSPI2( ADDR_LSB);    // send LSB of memory address
writeSPI2( data);        // send the actual data to be written
// send more data here to perform a page write
CSEE = 1;                // start actual EEPROM write cycle
```

注意，实际的 EEPROM 写循环是在 CS 再次变成高电平后才触发的。而且，在新指令执行之前，为完成写操作周期还必须有一个等待时间（Twc）（如存储设备数据表所说明的指标）。有两种方法可以用来确定存储器有足够的时间完成写命令。最简单的一种方法就是在写序列后插入固定的延时。延时的长度应该比存储设备数据表中给出的最大周期时间更长一些。

一种更好的方法是，在执行进一步的读/写命令之前，检查状态寄存器的内容，等待在线写入（WIP）标志位被清零（这和写允许位的复位操作同时发生）。这样，只需等待当前操作条件下存储设备所需的最少时间。

7.3.7 读存储器内容

要读回存储器的内容更加简单，下面的一小段代码可以实现必要的操作序列：

```
// send a Write command
CSEE = 0;                // select the Serial EEPROM
writeSPI2( SEE_READ);    // send command, ignore immediate data
writeSPI2( ADDR_MSB);    // send MSB of memory address
writeSPI2( ADDR_LSB);    // send LSB of memory address
data = writeSPI2( 0);    // send dummy, read data
// read more data here sequentially incrementing the address
CSEE = 1;                // terminate the read sequence, return to low power
```

读序列可以无限地执行，如果需要的话，可以连续读取整个存储器的内容，到达最后一个存储地址（0x7FFF）后，又会返回 0x0000 重新开始。

7.3.8 非易失性存储库

现在可以汇集一个小的函数库来访问 25LC256 串行 EEPROM。该函数库隐藏了所有的实现细节，如使用的 SPI 端口，特定的操作序列和定时操作。作为替代，仅使用两条基本命令就可实现向非易失性存储设备读取和写入整型数据。

下面使用 Project Wizard 和常规列表，生成一个新的项目，并命名为“NVM”。在生成新的源文件“nvm.c”后，可以将 SPI2 项目中的大部分定义复制到里面：

```
/*
** NVM Access Library
*/

#include <p24fj128ga010.h>

#include "NVM.h"

// I/O definitions for PIC24 + Explorer16 demo board
#define CSEE        _RD12          // select line for Serial EEPROM
#define TCSEE       _TRISD12       // tris control for CSEE pin

// peripheral configurations
#define SPI_MASTER  0x0122         // select 8-bit master mode, CKE=1, CKP=0
#define SPI_ENABLE 0x8000          // enable SPI port, clear status

// 25LC256 Serial EEPROM commands
#define SEE_WRSR    1              // write status register
#define SEE_WRITE   2              // write command
#define SEE_READ    3              // read command
#define SEE_WDI     4              // write disable
#define SEE_STAT    5              // read status register
#define SEE_WEN     6              // write enable
```

从上述项目中可以提炼出这几个功能：初始化代码、SPI2 写函数和状态寄存器读命令。每一个功能都可以作为独立的函数：

```
void InitNVM(void)
{
    // init the SPI peripheral
    TCSEE = 0;                         // make SSEE pin output
    CSEE = 1;                          // de-select the Serial EEPROM
    SPI2CON1 = SPI_MASTER;             // select mode
    SPI2STAT = SPI_ENABLE;             // enable the peripheral

}//InitNVM


int writeSPI2( int data)
{// send one byte of data and receive one back at the same time
    SPI2BUF = data;                    // write to buffer for TX
    while( !SPI2STATbits.SPIRBF);      // wait for transfer to complete
    return SPI2BUF;                    // read the received value
}//WriteSPI2


int ReadSR( void)
```

```
{// Check the Serial EEPROM status register
    int i;
    CSEE = 0;                          // select the Serial EEPROM
    WriteSPI2( SEE_STAT);              // send a READ STATUS COMMAND
    i = WriteSPI2( 0);                 // send/receive
    CSEE = 1;                          // deselect to terminate command
    return i;
} //ReadSR
```

要创建一个从非易失性存储器中读取整数的函数，首先要确认已经通过读状态寄存器正确地终止过去的命令（写命令）。两个连续字节的读命令用于读取出一个整数：

```
int iReadNVM( int address)
{ // read a 16-bit value starting at an even address

    int lsb, msb;

    // wait until any work in progress is completed
    while ( ReadSR() & 0x3);      // check the two lsb WEN and WIP

    // perform a 16-bit read sequence (two byte sequential read)
    CSEE = 0;                     // select the Serial EEPROM
    WriteSPI2( SEE_READ);         // read command
    WriteSPI2( address>>8);       // address MSB first

    WriteSPI2( address & 0xfe);  // address LSB (word aligned)
    msb = WriteSPI2( 0);          // send dummy, read msb
    lsb = WriteSPI2( 0);          // send dummy, read lsb
    CSEE = 1;
    return ( (msb<<8)+ lsb);
}//iReadNVM
```

最后，从过去的项目中抽取用来访问写允许锁存器的短程序代码，并添加页面写序列命令，完成写允许函数的创建，如下所列：

```
void WriteEnable( void)
{ // send a Write Enable command
    CSEE = 0;                          // select the Serial EEPROM
    WriteSPI2( SEE_WEN);               // write enable command
    CSEE = 1;                          // deselect to complete the command
}//WriteEnable


void iWriteNVM( int address, int data)
{ // write a 16-bit value starting at an even address

    int lsb, msb;

    // wait until any work in progress is completed
    while ( ReadSR() & 0x3);           // check the two lsb WEN and WIP

    // Set the Write Enable Latch
    WriteEnable();

    // perform a 16-bit write sequence (2 byte page write)
    CSEE = 0;                          // select the Serial EEPROM
    WriteSPI2( SEE_WRITE);             // write command
```

```
    WriteSPI2( address>>8);          // address MSB first
    WriteSPI2( address & 0xfe);      // address LSB (word aligned)
    WriteSPI2( data >>8);            // send msb
    WriteSPI2( data & 0xff);         // send lsb
    CSEE = 1;
}//iWriteNVM
```

还可以添加更多的函数来访问长整型和双长整型的数据，不过目前介绍的内容对于读者的学习已经够用。

注意“页写模式”（请参考25LC256存储器数据表上更多的技术细节）要求地址在2的幂次边界上对齐（本例中就是偶地址）。为了一致性，读函数也需要满足这个要求。

将程序代码保存在文件“nvm.c”中，并使用备忘录中三种方法中的任意一种方法，将文件添加到项目中。用户可以使用编辑器右键菜单，选择“Add to Project”或者右击项目窗口的“源文件”栏，选择“Add Files”，然后从当前项目目录中选择“NVM.c”文件。

为了使得从该模块中选择一些函数能被其他应用访问，创建一个新文件“NVW.h”，并插入如下的声明：

```
/*
** NVM storage library
**
** encapsulates 25LC256 Serial EEPROM
** as a NVM storage device for PIC24 + Explorer16 applications
*/

// initialize access to memory device
void InitNVM(void);

// 16-bit integer read and write functions
// NOTE: address must be an even value between 0x0000 and 0x7ffe
// (see page write restrictions on the device datasheet)
int  iReadNVM ( int address);
void iWriteNVM( int address, int data);
```

这里只列出了函数的初始化和整型数的读/写函数，隐藏了所有其他的实现细节。

通过右击项目窗口的头文件图标，从当前项目目录中选中“NVM.h”文件，将它添加到项目中。

7.3.9 测试新的NVM库

为了测试库的性能，现在生成一个包含以下几行代码的测试应用：不断读取内存地址内容（地址0x1234），并自增量，然后回写到内存中。

```
/*
** NVM Library test
*/

#include <p24fj128ga010.h>

#include "NVM.h"


main()
```

```
{
    int data;

    // initialize the SPI2 port and CS to access the 25LC256
    InitNVM();

    // main loop
    while ( 1)
    {
        // read current content of memory location
        data = iReadNVM( 0x1234);

        // increment current value
        Nop();                     // <-set brkpt here
        data++;

        // write back the new value
        iWriteNVM( 0x1234, data);
        //address++;

    } // main loop
} //main
```

将上面的文件保存为“NVMtest.c”，并添加到当前项目中。

调用 Build All 命令，读者可以观察到 MPLAB C30 编译器按顺序执行两个源文件（.c），然后连接器会连接目标代码，产生可执行的输出文件（.hex）。

这里使用 ICD2 作为调试工具来测试代码，因为 MPLAB SIM 不能准确地模拟 SPI 端口。不仅要保证调试器菜单（Debugger menu）已经选定，而且还要保证在“Project→Settings”中，特别是 MPLAB C30 连接器标签中，“Link for ICD2”选项也应被选定。（如图 7-8 所示。）

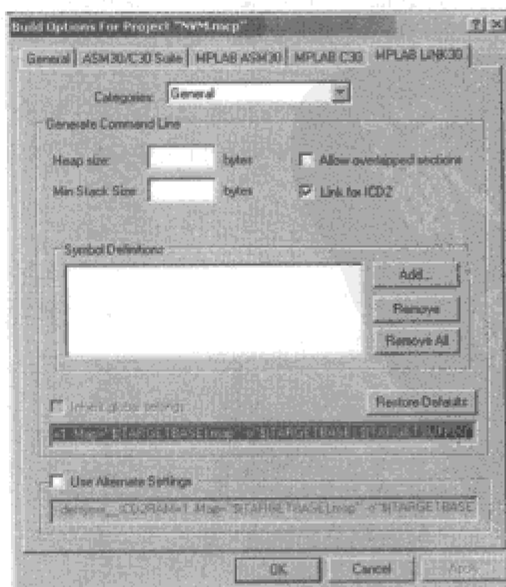


图 7-8　“Project→Build Options→MPLAB LINK30”标签

在使用 ICD2 调试器时，为了给 ICD2 自身保留一定的 RAM 地址（在存储器的最后），也为了避免同用户应用分配的存储空间发生冲突，以上的设置是必要的。

如果 Build All 命令顺利执行，那么程序代码就可以在设备上运行了。

(1) 将 `data` 添加到监视窗口，并在读命令后的代码行设置断点，测试 NVM 库的执行是否正确。

(2) 单击 Run 命令，可以观察到在第一次执行读命令后，程序停止了。

注意 `data` 的值，然后再次运行。可以发现它的值一直在增加，甚至当程序重启或者将演示板完全断电后再连接，都可以看到地址 0x1234 的内容被保留并且成功自增量。

小心——如果没有设置任何断点，主程序循环将一直不停地执行，测试程序就变成测试串行 EEPROM 的耐力了。实际上，循环程序按照一定的速度（大多数都取决于设备的实际 Twc 值），不断地改变地址 0x1234 的内容。最理想的情况（最大 Twc=5 ms）是每秒改变 200 次。或者，换而言之，EEPROM 的理论耐力极限（1 000 000 个周期）可达到 5 000 s，或者连续执行时间略少于 1.5 h。

7.4 飞后小结

本章简要介绍了如何使用 SPI 外部模块(以最简单的配置)来访问 25LC256 串行 EEPROM，即嵌入式控制应用中最常用的非易失性存储外设。希望在以后的使用中，小库模型可以为读者提供“更大”的存储容量（32KB）。

7.5 给 C 语言专家的提示

习惯于为大型工作站和个人电脑编写代码的 C 程序员，可能希望进一步开发包括可扩展复杂函数的函数库。本书的建议是先摒住气，然后深呼吸，一直数到十，特别是在向库函数里添加新的参数之前。在嵌入式控制世界中，传递更多的参数意味着要占用更多的栈空间，花费更多的时间在栈之间复制数据，并且会产生更大的输出代码。要保持库的简洁，以使其更易于测试和维护。这并不意味着适当的面向对象编程方法就不可以用。相反，上面的例子可以作为对象封装的一个例子，因为所有 SPI 接口和串行 EEPROM 内部工作的细节对用户来说是完全不可见的，用户看到的仅仅是一个连接普通存储设备的简单接口。

7.6 给汇编语言专家的提示

在开发上面代码的例子中，忽略了对访问速度的考虑，只是简单地把 SPI 模块看作最慢的可能操作。PIC24 SPI 外部模块支持外部的时钟系统，在目前的成品模型中，时钟系统最快可以达到 16 MHz。极少的外设可以在 3V 的供电下达到这种速度。特别是 25LC256 系列串行 EEPROM 在 2.5 V 到 4.5 V 的电源范围下，最大的时钟频率也只有 5 MHz。这意味着，同存储设备兼容的最快的 SPI 端口配置需要通过设置一级预分频器值（4：1）和二级预分频器值（1：1）来获得（这里有 16 MHz/4=4 MHz）。因此，连续的读命令能提供最大的吞吐量是 4 Mb/s 或者 512KB/s。以这样的速度，CPU 仍然能够在接收两个新字节数据之间的间隔中执行 32 条指令——尽管不足以执行复杂的计算，但是可以完成简单的数据传输任务。

7.7 给 PIC 微控制器专家的提示

对于大多数 PIC 微控制器的 SPI 接口，SPI 还有以下可选用途（由 SSP 和 MSSP 模块提供）：

- 可选的时钟极性；
- 可选的时钟边缘；
- 主/从模式操作。

此外，PIC24 的 SPI 接口模块还提供了很多新的功能，包括：

- 16 位传输模式；
- 数据输入采样相位选择；
- 帧传送模式；
- 帧同步脉冲控制（极性和边缘可选）；
- 增强模式（8 级发送和接收 FIFO）。

特别地，16 位传输模式可以在连续的读和/或页写操作期间使用，用以提升访问效率以及在访问 SPI 缓冲器时增加可用的周期数。不过在增强模式下，由于有 8 层 FIFO，因此可以节省相当一部分的 CPU 时间。在一个脉冲下，从 SPI 缓冲器中最多可以写入或读取 8 个字的数据（16 字节），这样就能在连续的脉冲之间，给 CPU 留下更多的时间来处理数据。

7.8 提示与技巧

如果读者想把重要的数据存放在一个外部的非易失性存储器中，那么可能需要增加一些对安全的考虑（包括硬件和软件）。从硬件角度来说，请确保以下各项。

- 在存储设备旁边设置足够的电源解耦电路（电容）。
- 在芯片片选线路上设置上拉电阻（10 kΩ），以避免在微控制器通电和重启时悬浮。
- 在 SCK 时钟线路上设置一个下拉电阻（10 kΩ），以避免在边缘扫描或者对其他电路板进行测试时，外围设备的时钟信号产生的干扰。
- 向微控制器提供干净而快速的上电或者断电信号，以保证复位操作的电源可靠。如果有需要，可增加一个外部的电源监视器（例如 MCP809 设备）。

还有很多的软件方法可以用来避免一些可能性极低的错误发生，例如一个程序缺陷或者公认的宇宙射线可能触发写操作。下面是一些具体的建议。

- 通电后，应避免立即读取，特别是改变 NVM 的内容。要等待几个毫秒，让电源稳定下来（这取决于应用）。
- 增加软件写使能标志位，要求在调用写子程序前先调用程序来将标志位置 1，有可能的话，还可以检查一些重要的入口条件。
- 增加栈等级计数器。库在栈中使用的每个函数，在进入的时候都应让计数器加 1，并在退出的时候减 1。如果计数器没有达到期望值，写程序应该拒绝执行。
- 有些程序员不愿意使用 NVM 的第一个存储地址（0x0000）和/或最后一个存储地址（0xffff），因为这些地址的内容更易于遭到损坏。
- 说实在的，每个重要的数据都应该有两个副本，执行两个分开的写子程序。如果每个副本都有简单的校验，那么就可以轻松地读取其中之一，以恢复受损的另一个。

7.9 练习

(1) 开发（带循环的）缓冲读和写函数。

(2) 使用新的 SPI16 位模式，加快基本的读写操作。

(3) 库里面的一些函数是锁定循环的，这会降低应用的整体性能。使用 SPI 端口中断，实现一个非块访问的库。

7.10 推荐书目

- Eady, F. (2004)
  Networking and Internetworking with Microcontrollers
  Newnes, Burlington, MA
  这本书有趣地介绍了嵌入式控制的串行通信。
- Buck, R. (1997)
  Flight of Passage: A Memoir
  Hyperion, New York, NY
  一次伟大的冒险，两个少年实现了横跨全国的飞行。

7.11 网上链接

- http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en010003
  使用上面的链接或者从 Microchip 的网站下载免费的工具“Total Endurance Software”。它可以帮助用户评估在实际应用条件下 NVM 设备的持续时间。它会明确给出在达到特定的目标错误率之前应用的擦写周期总数或者预期寿命年限。

第8章 异步通信

本章内容

- ▶UART 配置
- ▶发送和接收数据
- ▶测试串行通信程序
- ▶构建简单的控制库
- ▶测试 VT100 终端
- ▶使用串行端口作为调试工具
- ▶黑客帝国

如果读者具有无线通信的经验，那么就会明白无论是无线对讲机还是特定的民用波段广播其通信方式和手机通信都有很大的区别。首先，它是一种半双工系统，也就是说，当对方在说话的时候，自己是不能说话的。必须要耐心地听，等轮到自己的时候再高谈阔论，同时还要注意给其他人也留出发言的机会。这里要使用一个简单的口头上的握手系统来避免冲突和误会。

这也是航空上使用的方法，需要有一个确切的机制（也就是一系列的规则）来指明谁发言以及何时发言，乃至发言的内容和方式。此外，还需要几个分工明确的角色——例如空中交通调度员、飞行员、飞行基站和瞭望塔工作人员——他们以一种高效协作的方式共享通信媒介。

这种方式同样也可以引入到很多异步串行通信协议中来。虽然有些是全双工的，有些是半双工的；有些是多点通信的，而有些是点对点的，但是它们都需要合作并遵守基本的规则（标准），以保证通信正常，并且能够高效地使用通信媒介。

8.1 飞行计划

本章将回顾 PIC24 异步串行通信接口模块，以及 UART1 和 UART2，然后将开发一个基本的控制库，为后续项目的接口和调试提供便利。

8.2 飞前备忘录

除了诸如 MPLAB IDE、MPLAB C30 编译器和 MPLAB SIM 仿真器等常用的软件工具以外，本章还需要使用 Explorer16 演示板、MPLAB ICE2 电路内调试器和带有 RS232 串行端口（或者是可连接到 USB 适配器的串行口）的 PC。如果读者使用的是 Microsoft Windows 操作系统，那么还需要使用一个终端仿真程序，HyperTerminal 会是一个不错的选择。（“Start→Programs→Accessories→Communication→HyperTerminal”）。

8.3 飞行

UART 接口可能是嵌入式控制世界里最古老的接口。它的某些特性，可以追溯到第一台机械电传打字机的兼容性需求，这意味着它的一些技术已经有上百年的历史了。

另一方面，很难在今天的新电脑（特别是笔记本电脑）上找到异步串行端口。串行端口被称为“传统接口”，近几年，一股强大的力量推动着计算机制造商使用 USB 接口取代这种传统接口。尽管它们的流行程度在降低，而且 USB 接口优越的性能和特性显而易见，但是异步串行接口因其简单、低廉的特点而仍然广泛用于嵌入式应用中。

目前还在使用的异步串行应用主要有以下 4 大类。

(1) RS232 点对点连接：通常简称为“串行端口”，可用于终端、调制解调器和个人电脑，使用+12V/−12V 收发器。

(2) RS485（EIA-485）多点串行连接：用于工业应用，使用 9 位的字和特殊的半双工收发器。

(3) LIN 总线：为汽车外围应用而设计的低功耗、低电压总线。这里需要使用具有波特率自动检测功能的 UART。

(4) 红外无线通信：需要使用 38 kHz~40 kHz 的信号调制和光学收发器。

PIC24 的 UART 模块支持全部 4 种主要的应用，并新增了一些有趣的特性，如图 8-1 所示。

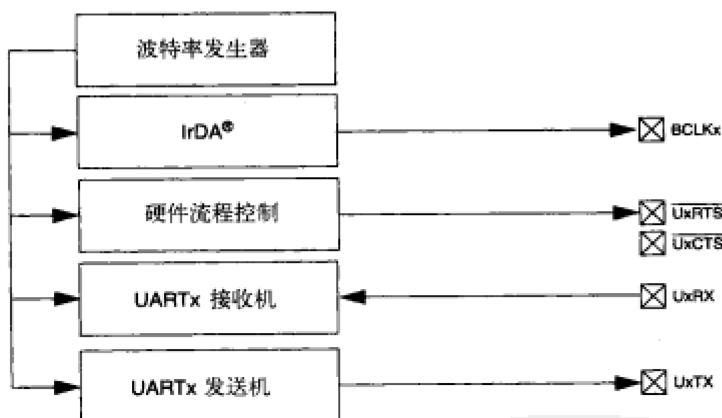


图 8-1 简化的 UART 模块图

为了说明外设 UART 的基本功能，这里使用了 Explorer16 演示板，其中 UART2 模块连接到 RS232 收发器和标准的 9 针 D 型母连接器。它也可以连接到任何的 PC 串行口，或者，如果在没有上面所说的“传统接口”时，也可以连接从 RS232 到 USB 的转换设备。无论是哪一种方式，Microsoft Windows 的 HyperTerminal 程序都能够以基本的配置完成与 Explorer16 演示板的数据交换。

第一步要定义传输参数。包括以下几种：

- 波特率；
- 数据位数；
- 奇偶校验位，如果有的话；
- 终止位数；
- 握手协议。

在演示实验中，选择一种方便快捷的模式——“115200，8，N，1，CTS/RTS”，即

- 波特率为 115 200；
- 8 位数据；
- 无奇偶校验；
- 1 位终止位；
- 握手协议使用 CTS 和 RTS 线。

8.3.1　UART 配置

使用“New Project Set-up”列表生成新项目“Serial”和源文件“serial.c”。在程序开始前，要添加一些有用的 I/O 定义，用来控制硬件握手信号线：

```
/*
** Asynchronous Serial Communication
** UART2 RS232 asynchronous communication demonstration code
*/

#include <p24fj128ga010.h>

// I/O definitions for the Explorer16
#define CTS         _RF12           // Clear To Send, input, HW handshake
#define RTS         _RF13           // Request To Send, output, HW handshake
#define     TRTSTRISFbits.TRISF13   // Tris control for RTS pin
```

当同 Windows 终端进行通信时，硬件握手尤其必要，因为 Windows 是一个多任务操作系统，应用程序有时会遇到可能导致数据丢失的长延时。这里使用一个 I/O 引脚作为输入（在 Explorer16 演示板上是 RF12）来检测终端是否准备就绪接收新字符（发送清零），还使用一个 I/O 引脚作为输出（在 Explorer16 演示板上是 RF13）来提示终端可以发送字符（发送请求）。

要设置波特率，就要使用波特率发生器（BREG2），一个源于外部时钟电路的 16 位计数器。根据设备数据表可以知道，在正常模式（BREGH=0）下，它支持 1：16 的分频，而在高速模式（BREGH=1）下，它的时钟支持 1：4 的分频。使用数据表上的一个简单公式，就可以让用户计算出理想的配置参数：

```
BREG2 =  (Fosc / 8 / baudrate) -1        ; for BREGH=1
```

在本次实验中，上面的公式可以转换成下面的形式：

```
BREG2 = (Fosc / 8 / 115,200) -1 = 33.7 where Fosc = 32MHz.
```

为了得出最佳的近似结果（毕竟这里使用的是 16 位整数），可使用下面的方程来计算实际的波特率和误差百分比：

```
Error = ((Fosc/ 8 / (BREG2 + 1)) - baudrate) / baudrate %
```

在四舍五入后得到 34，实际使用的波特率是 114 285 Bd，误差只有 0.7%，这属于可接受的误差范围。而对于 33，得到的波特率是 117 647，误差为 2.1%，这超出标准 RS232 端口可接受的误差范围（±2%）。

因此，常量 BRATE 可定义为：

```
#define BRATE    34           // 115200 Bd (BREGH=1)
```

另外的两个常量用于定义 UART2 主控制寄存器（U2MODE 和 U2STA）的初始值。

U2MODE 控制寄存器的初始值包括了 BREGH 位、终止位数和奇偶校验位设置，如图 8-2 所示。

```
#define U_ENABLE 0x8008          // enable UART, BREGH=1, 1 stop, no parity
```

高字节：

R/W-0	U-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
UARTEN	—	USIDL	IREN	RTSMD	—	UEN1	UEN0
位15							位8

低字节：

R/W-0 HC	R/W-0	R/W-0 HC	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WAKE	LPBACK	ABAUD	RXINV	BREGH	PDSEL1	PDSEL0	STSEL
位7							位0

图 8-2 U2MODE 控制寄存器

U2STA 控制寄存器的初始化，包括了对发送机的使能和对错误标志位的清零，如图 8-3 所示。

```
#define U_TX      0x0400         // enable transmission, clear all flags
```

高字节：

R/W-0	R/W-0	R/W-0	U-0	R/W-0 HC	R/W-0	R-0	R-1
UTXISEL1	UTXINV	UTXISEL0	—	UTXBRK	UTXEN	UTXBF	TRMT
位15							位8

低字节：

R/W-0	R/W-0	R/W-0	R-1	R-0	R-0	R/C-0	R-0
URXISEL1	URXISEL0	ADDEN	RIDLE	PERR	FERR	OERR	URXDA
位7							位0

图 8-3 UxSTA 控制寄存器

使用上面的常量定义，生成一个新的函数，对 UART2 控制寄存器、波特率发生器和用于握手的 I/O 引脚进行初始化。

```
void initU2( void)
{
    U2BRG   = BRATE;     // initialize the baud rate generator
    U2MODE  = U_ENABLE;  // initialize the UART module
    U2STA   = U_TX;      // enable the Transmitter
    TRTS    = 0;         // make RTS an output pin
    RTS     = 1;         // set RTS default status (not ready)
} // initU2
```

8.3.2 发送和接收数据

向串行端口发送一个字符可以分为以下 3 步。

(1) 确保终端（PC 上运行的 Windows HyperTerminal）已经就绪。检查发送（CTS）线路是否已被清零。CTS 是低电平有效的信号——当它为高电平时，用户需要耐心地等待。

(2) 确定 UART 已将以前的数据发送完毕。PIC24 的 UART 有 4 级的 FIFO 缓冲，因此用户至少需要等待直到最高级的 FIFO 被释放，换而言之，用户需要检查发送缓冲标志位 UTXBF 是否清零。

(3) 最后，向 UART 发送缓冲器（FIFO）传送新字符。

以上步骤的实现可以封装在一个简短的函数中：

```
int putU2( int c)
{
    while ( CTS);                // wait for !CTS, clear to send
    while ( U2STAbits.UTXBF);    // wait while Tx buffer full
    U2TXREG = c;
    return c;
} // putU2
```

要从串行端口接收字符，需要执行类似的步骤。

(1) 通过发出 RTS 信号（低电平有效），告知终端接收已准备就绪。

(2) 等待数据到达接收缓冲器，检查 URXDA 标志位。

(3) 从接收缓冲器（FIFO）读取字符。

同样地，以上的步骤也可以全部在一个简短的函数中实现：

```
char getU2( void)
{
    RTS = 0;                     // assert Request To Send !RTS
    while ( !U2STAbits.URXDA);   // wait for a new character to arrive
    return U2RXREG;              // read the character from the receive buffer
    RTS = 1;
}// getU2
```

8.3.3　测试串行通信程序

为了测试串行端口控制程序，下面编写一小段程序，用以实现串行口的初始化、发送提示信息，并在终端屏幕上显示终端键盘输入的字符：

```
main()
{
    char c;

    // 1. init the UART2 serial port
    initU2();

    // 2. prompt
    putU2( '>');

    // 3. main loop
    while ( 1)
    {

        // 3.1 wait for a character
        c = getU2();

        // 3.2 echo the character
```

```
        putU2( c);

    } // main loop

}// main
```

这里需要遵循下面的步骤。

(1) 首先建立项目，然后按照标准的列表打开 ICD2 调试器，并对 Explorer16 编程。

(2) 连接串行电缆到 PC（直接连接或者通过串口–USB 转换器），对 HyperTerminal 设置同样的通信参数：115 200、n、8、1，使用 COM 端口的 RTS/CTS 线。

(3) 单击 HyperTerminal Connect 连接按钮，开始终端仿真。

(4) 从调试（Debugger）菜单中选择“Run”，执行演示程序。注意：这里需要提醒读者，从现在起，在使用 UART 的时候，不要使用单步运行、设置断点或者 RunToCursor！请参阅 8.7 节，里面会给出详细的解释。

同时还要注意，如果 HyperTerminal 已经设置成显示每个发送的字符，那么用户将会看到两个字符！要禁止该功能，首先要单击 HyperTerminal 的“disconnect”（断开连接）按钮。然后选择“File→Properties”，并且在属性对话框里选中“Setting Pane Tab”（设置栏），如图 8-4 所示。现在还可以设置另外两个选项的内容，以使本章后面的操作更方便。

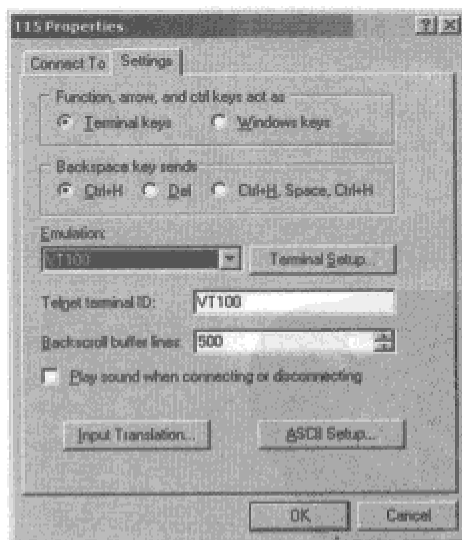


图 8-4 HyperTerminal 属性对话框，设置栏

(1) 选择 VT100 终端仿真模式，这样用户就可以使用更多的命令（由特殊的“escape”字符串触发），并对光标在终端屏幕上的位置进行更好的控制。

(2) 选择 ASCII Setup（ASCII 设置）完成终端配置。特别要注意“Echo typed characters locally”（本地响应键入字符）功能**没有**被选中（这将立即改进你的程序的效果）。(如图 8-5 所示。)

(3) 同时选中“Append line feeds to incoming line ends”选项。这样可以保证每次都能接收到 ASCII 回车符（'\r'），并自动插入换行符（'\n'）。

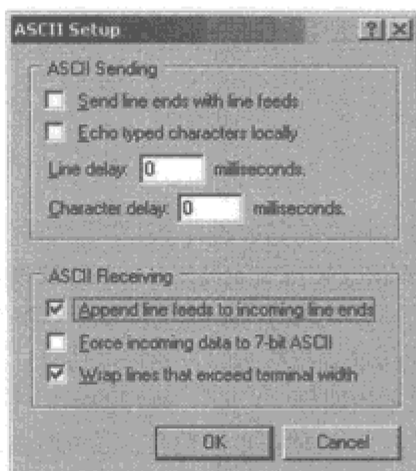


图 8-5 ASCII Setup 对话框

8.3.4 建立简单的控制库

为了方便以后的项目，现在要将上面的演示项目转换成适当的终端控制库，只需多添加两个函数就能实现：一个函数用于打印整个（遇到零就终止）字符串，另一个函数用于输入整行文本。正如读者可能想到的，一个简单的字符串打印程序是：

```
int putsU2( char *s)
{
    while( *s)              // loop until *s == '\0', end of string
        putU2( *s++);       // send the character and point to the next one
} // putsU2
```

这个简单的循环只是不断地调用 putU2 函数，将字符串中的字符一个接一个地发送到串行端口。

将文本字符串从终端（控制台）读取到字符串缓冲器也是同样的简单，不过要先保证字符串的长度没有超出缓冲器的大小（用户键入的字符串不能太长），并且在字符串结束的时候，要将回车符转换成正确的'\0'字符。

```
char *getsnU2( char *s, int len)
{
    char *p = s;            // copy the buffer pointer
    do{
        *s = getU2();       // wait for a new character

        if ( *s=='\r')      // end of line, end loop
            break;
        s++;                // increment buffer pointer
     len--;
    } while ( len>1 );      // until buffer full
    *s = '\0';              // null terminate the string

    return p;               // return buffer pointer
} // getsnU2
```

事实表明，这个函数使用起来是很麻烦的。由于不能显示输入的字符，因此用户也检查不出错误。只要存在很小的输入错误，整行数据就都必须重新输入。如果读者老是输入错误，那么键盘上点击率最高的按键就会是退格键。一个更好的 getsnU2 函数应该包含字符回显和退格键的基本编辑功能。这只需要多添加两行代码就可以实现。回显将在字符接收后马上发生。经过译码的退格字符（ASCII 码定义为 0x8）被用来将缓冲器指针向后移动一个字符的位置（只要不是位于行首）。同时还要输出特殊的字符序列，在屏幕上显示前一个字符的移除。

```
char *getsnU2( char *s, int len)
{
    char *p = s;            // copy the buffer pointer
    int cc = 0;             // character count
    do{
        *s = getU2();       // wait for a new character
        putU2( *s);         // echo character

        if (( *s==BACKSPACE)&&( s>p))
        {
            putU2( ' ');      // overwrite the last character
            putU2( BACKSPACE);
            len++;
            s--;            // back the pointer
            continue;
        }
        if ( *s=='\n')      // line feed, ignore it
            continue;
        if ( *s=='\r')      // end of line, end loop
            break;
        s++;                // increment buffer pointer
        len--;
    } while ( len>1 );      // until buffer full

    *s = '\0';              // null terminate the string

    return p;               // return buffer pointer
} // getsnU2
```

将所有的函数都放在独立的文件“conU2.c”中。然后生成一个小的头文件“conU2.h”，以确定哪个函数（原型）和哪些常量对于外部世界是可见和公开的。

```
/*
** CONU2.h
** console I/O library for Explorer16 board
*/

// I/O definitions for the Explorer16
#define CTS         _RF12       // Clear To Send, input, HW handshake
#define RTS         _RF13       // Request To Send, output, HW handshake
#define BACKSPACE 0x8           // ASCII backspace character code

// init the serial port (UART2, 115200@32MHz, 8, N, 1, CTS/RTS )
void initU2( void);
```

```
// send a character to the serial port
int putU2( int c);

// wait for a new character to arrive to the serial port
char getU2( void);

// send a null terminated string to the serial port
int putsU2( char *s);

// receive a null terminated string in a buffer of len char
char * getsnU2( char *s, int n);
```

8.3.5 测试 VT100 终端

由于已经使能 VT100 终端仿真模式（请参阅上面的 HyperTerminal 设置），现在只需要几个命令就可以更好地控制终端屏幕和光标位置，如以下命令：

- ❑ clrscr，终端屏幕清零；
- ❑ home，光标归位，回到屏幕的左上方。

上面命令的执行需要发送"转义序列"（在 ECMA-48 标准中已有定义，请参阅 ISO/IEC6429 和 ANSI X3.64），即 ANSI 转义码。它们都是以字符"ESC（ASCII 码 0x1b）"和"["（左方括弧）开始的：

```
// useful macros for VT100 terminal emulation
#define clrscr() putsU2( "\x1b[2J")
#define home()   putsU2( "\x1b[1,1H")
```

为了测试控制库，下面编写一小段程序，实现以下的功能。

(1) 串行端口初始化。
(2) 终端屏幕清空。
(3) 发送欢迎信息/横幅。
(4) 发送提示字符。
(5) 读取整行文本。
(6) 打印新的文本行。

在新文件"CONU2test.c"中保存以下的代码：

```
/*
** CONU2 Test
** UART2 RS232 asynchronous communication demonstration code
*/

#include <p24fj128ga010.h>
#include "conU2.h"

#define BUF_SIZE 128

main()
{
    char s[BUF_SIZE];

    // 1. init the console serial port
```

```
    initU2();

    // 2. text prompt
    clrscr();
    home();
    putsU2( "Learn to fly with the PIC24!");

    // 3. main loop
    while ( 1)
    {
          putU2(">");     // prompt

      // 3.1 read a full line of text
          getsnU2( s, BUF_SIZE);

     // 3.2 send a string to the serial port
          putsU2( s);

     // 3.3 send a carriage return
     putU2('\r');

    } // main loop

}// main
```

共操作步骤如下。

(1) 使用“New Project”列表生成新项目，并加入“conU2.h”、“conU2.c”和“conU2test.c”。

(2) 使用 ICD2 列表连接 ICD2 调试器，并对 Explorer16 演示板编程。

(3) 测试刚刚完成的控制库的编辑能力。

8.3.6 使用串行端口作为调试工具

当有了一个可以通过串行端口向控制器发送和接收数据的小函数库时，读者就掌握了一个新的有用的调试工具。读者可以调用打印函数，在终端上显示重要变量和其他诊断信息的内容。并且可以轻松地将输出信息格式化为最易读的格式。此外还可以加入设置参数的输入函数，以更好地测试代码或者在必要时使用输入函数来暂停程序执行，让读者有时间阅读诊断输出信息。这是最老的调试工具，从第一台计算机问世以来，它一直是有效的调试方法。

8.3.7 黑客帝国

下面以一个更有趣的内容来结束本章，首先要生成一个新的演示项目“matrix.c”。该项目旨在通过向终端发送大量的文本和性能度量来测试串行端口和 PC 终端仿真的速度。唯一的问题是，程序缺少一个大容量的存储设备，以读取其中的内容并发送给终端。因此，最好的解决办法就是使用伪随机数发生器去“生成”大容量的内容信息。库“stdlib.h”实际上提供了方便的 rand() 函数，可以返回 0 到 MAX_RAND（在“limits.h”文件中定义的常量，对于 MPLABC20 来说是 32 767）之间的正数。

使用“remainder of”运算符，可以将输出结果压缩在任意更小的整数范围内，并从 ASCII 码集中只产生可打印的字符子集。例如下面的语句只生成 33 到 127 范围内的字符：

```
putU2( 33 + (rand()%94));
```

为了生成一个更有吸引力和趣味性的输出（尤其是在读者看过电影《黑客帝国》的情况下），我们将以列而不是行的形式呈现（随机的）内容。当不停地刷新屏幕时，将使用随机数发生器来改变每一列的内容和“长度”。

```
/*
**  The Matrix
**
*/
#include <p24fj128ga010.h>

#include "CONU2.h"
#include <stdlib.h>

#define COL     40
#define ROW     23

#define DELAY 3000

main()
{
    int v[40];  // vector containing length of each string
    int i,j,k;

    // 1. initializations
    T1CON = 0x8030;   // TMR1 on, prescale 256, Tcy/2
    initU2();         // initialize the console (115200, 8, N, 1, CTS/RTS)
    clrscr();         // clear the terminal (VT100 emulation)
    getU2();          // wait for one character to randomize the sequence
    srand( TMR1);

    // 2. init each column length
    for( j =0; j<COL; j++)
            v[j] = rand()%ROW;

    // 3. main loop
    while( 1)
    {
        home();

        // 3.1 refresh the screen with random columns
        for( i=0; i<ROW; i++)
        {
            // refresh one row at a time
            for( j=0; j<COL; j++)
            {
                // print a random character down to each column length
                if ( i < v[j])
                    putU2( 33 + (rand()%94));
                else
                    putU2(' ');
                putU2( ' ');
            } // for j
```

```
            pcr();
        } // for i

        // 3.2 randomly increase or reduce each column length
        for( j=0; j<COL; j++)
        {
            switch ( rand()%3)
            {
                case 0: // increase length
                        v[j]++;
                        if (v[j]>ROW)
                            v[j]=ROW;
                        break;

                case 1: // decrease length
                        v[j]--;
                        if (v[j]<1)
                            v[j]=1;
                        break;

                default:// unchanged
                        break;
            } // switch
        } // for

    } // main loop
} // main
```

抛开机器的性能，观察这些代码的运行就十分有趣。不过它还是太快了——实际上，读者需要加入一个小小的延时循环（在程序段 3.1 中插入），那样眼睛看起来会更舒服：

```
// 3.1.1 delay to slow down the screen update
TMR1 =0;
while( TMR1<DELAY);
```

注意：下次记得带上眼药水。

8.4 飞后小结

本章在介绍作为 RS232 串行口的 UART 模块的基本功能时，开发了一个小的控制台 I/O 库，并且将 Explorer16 和 VT100（仿真）终端（Windows HyperTerminal）连接在一起。在后面的章节中，将利用该函数库作为更多高级飞行/项目的新型调试工具以及潜在的用户界面。

8.5 给 C 语言专家的提示

我相信，读者现在应经开始琢磨如何使用“stdio.h”里更高级的函数库来定向输出到 UART2 外设。其实，只要简单地替换其中一个主要的库函数“write.c”就可以了：

```
/*
** write.c
** replaces stdio lib write function
**
*/
```

```
#include <p24fj128ga010.h>
#include <stdio.h>
#include "conu2.h"

int write(int handle, void *buffer, unsigned int len)
{
    int i, *p;
    const char *pf;

    switch (handle)
    {
    case 0: // stdin
    case 1: // stdout
    case 2: // stderr
        for (i = len; i; --i)
            putU2( *(char*)buffer);
        break;
    default:
        break;
    } // switch
    return(len);
} // write
```

将上面的代码保存在当前项目目录下的“write.c”文件中，并加入到项目的源文件列表中。

从现在开始，连接器会连接和调用任何的“stdio.h”库函数，所生成的标准流（stdin、stdout 和 stderr）将重定向到 UART2。

注意，用户还是要正确地初始化 UART，并且“conu2.c”文件也必须包含在项目源文件中。

8.6 给 PIC 微控制器专家的提示

嵌入式控制设计师迟早是要面对 USB 总线的。尽管现在使用转接器（将串口转换到 USB 接口）还算是一个可行的解决方案，然而 USB 总线卓越的性能和兼容性迟早会让你将你的设计转向 USB 接口。一些 8 位 PIC 微控制器模型已经将 USB 串行接口引擎（SIE）作为标准的通信接口。Microchip 公司提供了一个免费的 USB 软件包，包括驱动程序和大部分常见应用的分类解决方案。其中，有一个叫作“通信设备类”（或者 CDC）的应用，使用它可以实现 USB 与 PC 应用之间的无缝连接，哪怕 HyperTerminal 也不测试它们的差别。最重要的是，用户不需要写和/或安装任何特殊的 Windows 驱动程序。当用 C 编写程序的时候，如果不需要设置通信参数，那么用户根本察觉不到其中的区别。使用 USB 接口的时候，不用设置波特率，没有奇偶校验，没有端口号的选择（错误），而通信速度却还要高出很多。

8.7 提示与技巧

关于 ICE 上的 ICD2 和 UART

正如在 8.3.3 节的一个练习中看到的，当允许和使用 UART 来同 HyperTerminal 程序发送和接收数据时，使用单步运行是非常糟糕的做法。在看到 HyperTerminal 程序工作失常和/或简单

停滞，毫无缘由地放过收到的数据时，用户的神经都快崩溃了。为了找出问题的症结所在，首先要深入理解 MPLAB ICD2 的电路内调试操作。在单步运行模式下或者遇到断点时，当执行完一条指令后，ICD2 调试器不仅会终止 CPU 的运行，还会“冻结”所有的外设。这是个突如其来的“霜冻”——甚至还没来得及传送一个时钟脉冲。若发生在正处于传送过程的 UART 外设时，串行输出线（TX）也会被冻结在当前状态。如果在那一瞬间，某一位信息正要发送，特别当它是“1”时，TX 线就会永远保持在“断开”状态。

另一方面，HyperTerminal 程序可以检测出这个“断开”状态并解释为一个线路错误，也就是假设连接丢失和连接断开。因为 HyperTerminal 是一个非常“低级”的程序，它不会告诉用户现在正在发生什么事……它不会发出任何的声音、错误信息等——它只会锁定!

如果用户发现了潜在的错误，那就不是什么大的问题。当使用 ICD2 重启程序时，只需要记住先单击“HyperTerminal Disconnect（断开）”按钮，然后再按“Connect（连接）”按钮，所有的操作都将恢复正常。

8.8 练习

编写带缓冲“I/O”功能（使用中断）的控制台库，使得对程序执行（和调试）的影响最小。

8.9 推荐书目

- Eady, F. (2005)
  Implementing 802.11 with Microcontrollers:
  Wireless Networking for Embedded Systems Designers
  Newnes, Burlington, MA
  Fred 将他的幽默和经验运用在了嵌入式编程中，让无线网络变得更简单。
- Axelson，J.(1999)
  USB Complete, 3rd ed.
  Lakeview Research, Madison, WI
  Jan 的书已经是第三版了。每一次她都加入了更多的材料，而且总是保持内容的简单。

8.10 网上链接

- http://en.wikipedia.org/wiki/ANSI_escape_code
  这个链接给出了实现 VT100HyperTerminal 仿真需要的 ANSI 转义码的全部列表。

第9章 玻璃护航

本章内容

- ▶HD44780 控制器的兼容性
- ▶并行主控制端口
- ▶LCD 模块控制的 PMP 配置
- ▶访问 LCD 显示的小函数库
- ▶高级 LCD 控制

过去，从最小的单引擎 Cessna 飞机到大型超音速 Concord 飞机，它们的驾驶舱都充斥着形如蒸汽仪的大型球状仪器。六种主要的仪器，因为总是按相同的顺序放置，所以被亲切地叫作“六件套”。不过，读者下次走进民用客机，记得寻找机会偷偷地瞥一眼驾驶舱。尽管里面还是有一大堆的按钮和开关，然而可以看出来，飞行员前方的控制台已经发生了巨大的变化。那是一块（或者两块）很大的平面玻璃。飞行员将这个改进叫做“玻璃”，尽管他们大多数都不知道里面的硅片超出了他们的想象多少倍。这就是驾驶舱的数字革命，而且只是在近年才发生的。

“玻璃”的后面是无数功能强大的微处理器在努力地把尽可能多的信息转换到简单的、直观的、亲切的界面上。全球定位系统（GPS）技术得益于这项技术革命，而现在每个飞机制造商都会为新的模块提供几个高级的“玻璃”驾驶舱。其中一些只是为了增加新飞机的销售量而作的投机，后来就激发了整个行业生产“玻璃驾驶舱”的热情。

不过，这一类的飞机并不是飞行学员在学习初期可以驾驭的。新型的飞机走进学堂还需要一些时间，不过这也只是时间的问题——玻璃护航就在眼前。

嵌入式世界也大量运用了玻璃，例如 LCD 显示。下面就开始探索基本的 LCD 接口。

9.1 飞行计划

在本章中，将会介绍一种小型又便宜的 LCD 显示模块的接口。这也是一个学习和使用并行主控制端口（PMP）的绝好机会。它是 PIC24 微控制器新增的一个多用途并行接口。

9.2 飞前备忘录

除了诸如 MPLAB IDE、MPLAB C30 编译器和 MPLAB SIM 仿真器等常用的软件工具以外，本章还需要用到 Explorer16 演示板和 MPLAB ICD 电路内调试器。

9.3 飞行

Explorer16 演示板可以提供三种不同类型的点阵文字数字 LCD 显示模块和一种图形 LCD 显示模块。在默认状态下，它带有简单的“2 行 16 字符”显示和 3V 的文字数字 LCD 模块（Tianma TM162JCAWG1），同工业标准级的 HD44780 控制器兼容，如图 9-1 所示。这类 LCD 模块是完

整的显示系统，包括 LCD 玻璃基板、行列多路驱动器、供电电路和智能控制器，它们全部使用玻璃基极晶片（COG）技术进行集成。由于有了这种高级的集成技术，使得点阵显示的控制电路变得很简单。LCD 模块的接口只需要一个使用 11 个 I/O 引脚的 8 位简单并行总线，而不是使用上百个引脚来驱动和控制每个行列的像素。

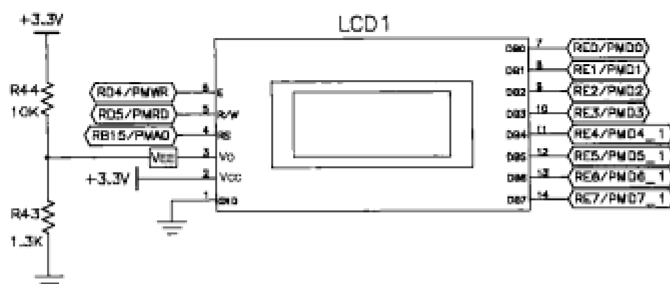


图 9-1 默认的文字数字 LCD 模块连接

特别是在文字数字模式，用户可以直接将 ASCII 字符代码放入 LCD 模块控制器的 RAM 缓冲器（DDRAM）中。输出的图像由一个使用 5×7 像素网格来生成每个显示字符的集成字符发生器（表）产生，如图 9-2 所示。表中还特别地添加了扩展的 ASCII 码集，在某种意义上还包括了日本汉字字符和一些常用符号集。字符发生表大多数都在显示控制 ROM 中实现，通过修改/创建可访问备用的内部 RAM 缓冲器（CGRAM）的新字符（有些模型多达 8 个），许多显示模型都可以对显示字符集进行扩展。

图 9-2 HD44780 兼容的 LCD 显示控制器使用的字符发生表

9.3.1 HD44780 控制器的兼容性

正如前面介绍的，Explorer16 演示板使用的 2×16 LCD 模块是市面上常见的 LCD 显示模块之一。其主要配置有 1 条线到 4 条线的 4 个等级，分别支持 8、16、20、32 个字符，其最多可

支持40个字符，这与今天已成为工业标准的HD44780芯片集是兼容的。

HD44780的兼容性是指集成控制器仅包含两个独立寻址的寄存器，一个用于ASCII数据，另一个用于命令。如表9-1和表9-2所示的标准指令集可以用于设置和控制显示。

表9-1　HD44780指令集

指令	代码										描述	执行时间
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
清除显示	0	0	0	0	0	0	0	0	0	1	清除显示内容，光标归位（地址0）	1.64 ms
光标归位	0	0	0	0	0	0	0	0	1	*	光标归位（地址0）。同时显示回到起始位置。DDRAM内容不变	1.64 ms
进入模式设置	0	0	0	0	0	0	0	1	I/D	S	设置光标移动方向（I/D），指定显示的移动（S）。这些操作在数据读/写期间执行	40 us
显示控制开/关	0	0	0	0	0	0	1	D	C	B	设置：打开/关闭所有显示（D）、光标开/关（C）、闪烁光标位置字符（B）	40 us
光标/显示的移位	0	0	0	0	0	1	S/C	R/L	*	*	设置：光标移动或者显示移位（S/D）、移动方向（R/L）。DDRAM内容保持不变	40 us
功能设置	0	0	0	0	1	DL	N	F	*	*	设置接口数据长度（DL）、显示行数（N）和字符样式（F）	40 us
CGRAM地址设置	0	0	0	1	CGRAM地址						设置CGRAM地址。CGRAM数据在设置后被收发	40 us
DDRAM地址设置	0	0	1	DDRAM地址							设置DDRAM地址。DDRAM数据在设置后被收发	40 us
读取忙碌标志位和地址计数器	0	1	BF	CGRAM/DDRAM地址							忙碌标志位（BF）说明正在执行内部操作。该指令读取BF和CGRAM或DDRAM地址计数器的内容（基于前一个指令）	0 us
CGRAM或DDREAM写入	1	0	写数据								向CGRAM或DDRAM写入数据	40 us
CGRAM或DDREAM读取	1	1	读数据								从CGRAM或DDRAM读取数据	40 us

有了这些共通性，在Explorer16演示板上用来驱动LCD的代码都可以立即应用到其他HD44780的兼容文字数字LCD显示模块上。

表 9-2 HD44780 指令位

位名称	设置/状态	
I/D	0=光标位置后退	1=光标位置前进
S	0=无显示转换	1=显示转换
D	0=显示关闭	1=显示打开
C	0=不显示光标	1=显示光标
B	0=光标不闪烁	1=光标闪烁
S/C	0=移动光标	1=移位显示
R/L	0=向左移	1=向右移
DL	0=4 位接口	1=8 位接口
N	0=1/8 或 1/11 周期（1 行）	1=1/16 周期（2 行）
F	0=5×7 点阵	1=5×10 点阵
BF	0=接收指令	1=正在处理内部操作

9.3.2 并行主控制端口

所有这些显示模块所共有的 8 位总线的简单结构值得关注。除了 8 条双向数据线（在“半位”模式下可以减少到 4 条 I/O 线）外，还包括以下几种控制线：

- 一条使能选通线（E）；
- 一条读/写选择线（R/W）；
- 一条用于寄存器选择的地址线（RS）。

通过控制 PORTE 和 PORTD 引脚来控制 11 条 I/O 线以实现总线序列是很简单的，不过这里要顺便介绍 PIC24 结构的一个新的外设功能：并行主控端口（PMP）。PIC24 系列的设计师使用这个新的可寻址并行端口可以加快访问大量常用外部并行设备的速度，包括模数转换器、RAM 缓冲器、ISA 总线兼容性接口、LCD 显示模块，甚至是硬盘和 CompactFlash 存储卡。

读者可以将 PMP 想象成是添加到 PIC24 结构上的一类灵活的 I/O 总线，但它却不会干扰（或者降低）24 位程序存储器总线和 16 位数据存储器总线的操作。PMP 能提供：

- 8 位或者 16 位的双向数据通道；
- 最大 64KB 的地址空间（16 条地址线）；
- 6 条附加的选通/控制线——使能、地址锁定、读、写以及两条片选线。

PMP 也可以配置为受控（从控制）模式，作为更大的微处理器/微控制器系统的可寻址外设。

总线的读写序列都是可编程的，因此用户不仅可以设置与目标总线相匹配的奇偶校验和控制信号，还可以根据接入外设的速度设定合适的时序信号。

9.3.3 LCD 模块控制的 PMP 配置

像 PIC24 的所有外设一样，PMP 的特性由特定的控制寄存器来设置。首先是 PMCON，读者可以发现它和其他所有模块的 xxCON 寄存器有很多相似的控制位，如图 9-3 所示。

高字节：

R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PMPEN	—	PSIDL	ADRMUX1	ADRMUX0	PTBEEN	PTWREN	PTRDEN
位 15							位 8

低字节：

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
CSF1	CSF0	ALP	CS2P	CS1P	BEP	WRSP	RDSP
位 7							位 0

图 9-3 PMCON 控制寄存器

不过，这次要初始化的控制寄存器有点多，包括：PMMODE、PMADDR、PMSTAT、PMPEN 和 PADCFG1。它们都有很强大的功能并且需要小心地设置。这里不详细介绍每个寄存器的细节，而只是列出 LCD 模块接口需要的关键配置。

- PMP 使能。
- 完全多路输出接口（使用独立的数据和地址线）。
- 选通信号使能（RD4）。
- 读信号使能（RD5）。
- 选通操作高电平有效。
- 读操作高电平有效，写操作低电平有效。
- 同一个引脚（RD5）上使用可读写的主控模式。
- 8 位总线接口（使用 PORTE 引脚）。
- 只需要 1 个地址位，因此选择由 PMA0（RB15）和 PMA1 组成的最小化配置。

同时，考虑到典型的 LCD 模块是一个速度非常慢的设备，最好选择较长的定时，即允许在读写序列的每个阶段中插入最大数量的等待状态。

- 4×Tcy 在读/写之前的等待数据建立。
- 15×Tcy 在 R/W 和使能之间等待。
- 4×Tcy 在使能之后等待数据建立。

9.3.4 访问 LCD 显示的小函数库

使用“New Project”列表生成新项目和源文件。

首先要编写的是 LCD 初始化代码。自然地，以 PMP 端口主要控制寄存器的初始化作为开始：

```
void LCDinit( void)
{
    // PMP initialization
    PMCON = 0x83BF;          // Enable the PMP, long waits
    PMMODE = 0x3FF;          // Master Mode 1
    PMPEN = 0x0001;          // PMA0 enabled
```

经过这些步骤，就可以首次与 LCD 模块通信了，然后用户可以应用制造商提供的标准 LCD 初始化序列。该初始化序列在时序上必须准确，请参阅 HD44780 指令集。在 LCD 模块处理完自身内部的初始化（上电复位）序列的 30 ms 后，才能进行这个初始化。为了简单和安全起见，用户应该在 LCD 模块初始化函数里加入强制的延时代码，这里将使用 Timer1 对所有的子程序

进行简单而精确的定时循环：

```
// init TMR1
T1CON = 0x8030;              // Fosc/2, prescaled 1:256, 16us/tick

// wait for >30ms
TMR1 = 0; while( TMR1<2000);    // 2000 x 16us = 32ms
```

为了方便起见，这里还定义了两个常量，使得下面的代码可读性更好：

```
#define LCDDATA 1              // RS = 1 ; access data register
#define LCDCMD  0              // RS = 0 ; access command register
#define PMDATA  PMDIN1         // PMP data buffer
```

要向 LCD 模块发送命令，首先要选择命令寄存器（设置地址 PMA0=RS=0）。然后将目标命令放在 PMP 数据输出缓冲器，开始一个 PMP 写序列：

```
PMADDR = LCDCMD;               // select the command register (ADDR = 0)

PMDATA = 0b00111000;           // function set: 8-bit interface, 2 lines, 5x7
```

PMP 将执行完整的总线写序列，如下所示。

(1) 地址出现在 PMP 地址总线（PMA0）。

(2) PMDATA 的内容出现在 PMP 数据总线上（PMD0~PMD7）。

(3) 经过 4×Tcy 的等待，R/W 信号转换成低电平（RD5）。

(4) 经过 15×Tcy 的等待，选通使能设置成高电平（RD4）。

(5) 经过 4×Tcy 的等待，选通使能变成低电平，PMDATA 从总线上移除。

要注意，上面的序列在 PIC24 初始化后，还需要大于 20 × Tcy 或者 1.25 μs 的时间。换而言之，PMP 在 PIC24 已经执行完 20 条或更多条指令后，仍然在执行以上的序列内容。由于需要一段相当长的时间（>40 us）来等待 LCD 模块执行命令，因此用户这次不需要考虑 PMP 完成指令的时间问题：

```
TMR1 = 0; while( TMR1<3);    // 3 x 16us = 48us
```

然后类似地，执行剩下的 LCD 模块初始化操作：

```
PMDATA = 0b00001100;         // display ON, cursor off, blink off
TMR1 = 0; while( TMR1<3);    // 3 x 16us = 48us

PMDATA = 0b00000001;         // clear display
TMR1 = 0; while( TMR1<100);  // 100 x 16us = 1.6ms

PMDATA = 0b00000110;         // increment cursor, no shift
TMR1 = 0; while( TMR1<100);  // 100 x 16us = 1.6ms
```

LCD 模块初始化之后，事情就变得更加简单了，定时循环也不再必要了，因为现在可以使用 LCD 模块读忙碌标志位命令了。它将告诉用户集成的 LCD 模块控制器是否已经完成最后一条指令，并且准备好接收和执行下一条命令。为了读取 LCD 状态寄存器的忙碌标志位，需要 PMP 执行总线读序列。这包括两个步骤：首先读取（并删除）PMP 数据缓冲器的内容来初始化读序列，当 PMP 序列完成时，数据缓冲器将包含从总线上读取的数据，此时用户再次从 PMP

数据缓冲器中读取内容。可是用户怎么知道 PMP 读序列是否完成了呢？很简单——用户可以通过检查 PMSTAT 控制寄存器中的 PMP 忙碌标志位来确定。

总的来说，要检查 LCD 模块的忙碌标志位，需要先检查 PMP 的忙碌标志位，发送读命令，再次等待 PMP 忙碌标志位信息，最后将访问 LCD 模块状态寄存器的包括 LCD 忙碌标志位在内的内容。

将寄存器地址作为参数传递给读函数，这里有一个更加常见的函数可以用来读取 LCD 状态寄存器和数据寄存器，其代码如下：

```
char LCDread( int addr)
{
    int dummy;
    while( PMMODEbits.BUSY);      // wait for PMP to complete previous commands
    PMADDR = addr;                // select the command address
    dummy = PMDATA;               // initiate a read cycle, dummy read
    while( PMMODEbits.BUSY);      // wait for PMP to complete the sequence
    return( PMDATA);              // read the status register
} // LCDread
```

LCD 模块状态寄存器有两个信息：LCD 忙碌标志位和 LCD RAM 指针当前值。通过两个简单的宏命令就可以分离这两个信息：LCDbusy()和 LCDaddr()，还有第三个宏命令可用来访问数据寄存器——getLCD()：

```
#define LCDbusy() LCDread( LCDCMD) & 0x80
#define LCDaddr() LCDread( LCDCMD) & 0x7F
#define getLCD()  LCDread( LCDDATA)
```

使用函数 LCDbusy()，就可以生成向 LCD 模块写入数据或者命令的函数：

```
void LCDwrite( int addr, char c)
{
    while( LCDbusy());
    while( PMMODEbits.BUSY);      // wait for PMP to be available
    PMADDR = addr;
    PMDATA = c;
} // LCDwrite
```

该函数库还可以补充下面的宏：

- putLCD() 将 ASCII 数据发送到 LCD 模块：

```
#define putLCD( d)  LCDwrite( LCDDATA, (d))
```

- LCDcmd() 将命令发送到 LCD 模块：

```
#define LCDcmd( c)  LCDwrite( LCDCMD, (c))
```

- LCDhome() 将光标返回到第一行的第一个字符：

```
#define LCDhome()   LCDwrite( LCDCMD, 2)
```

- LCDclr() 清除所有的显示内容：

```
#define LCDclr()    LCDwrite( LCDCMD, 1)
```

最后，为了使用方便，可以加入函数 putLCD()，它会向显示模块发送全空的终止字符串：

```
void putsLCD( char *s)
{
    while( *s)
        putLCD( *s++);
} //putsLCD
```

现在，增加一个简短的主函数，让这些函数工作起来：

```
main( void)
{
    // initializations
    LCDinit();

    // put a title on the first line
    putsLCD( "Flying the PIC24");

    // main loop, empty for now
     while ( 1)
     {
     }
} // main
```

如果项目建立并且使用 ICD2 调试器对 Explorer16 演示板编程之后一切正常的话，那么读者将会很有成就感地看到 LCD 显示屏的第一行显示出了标题字符串。

9.3.5 高级 LCD 控制

如果读者觉得上面的内容还不够复杂，没有满足感，下面再介绍一个更有趣味和挑战性的工作。

在介绍 HD44780 兼容文字数字 LCD 模块的时候，曾经提到过如何显示由模块控制器使用 ROM 中的一个表（字符发生器）来生成显示内容。不过，同时也提到了可以使用附加的 RAM 缓冲器（CGRAM）的扩展字符集。通过写入 CGRAM 的方法，可以生成 5×7 的字符图案，来创造出新的字符和小的图形元素。

让 Explorer16 LCD 模块来显示字符集中的一架小飞机怎么样？

首先需要使用函数将 LCD 模块的 RAM 缓冲器指针指向 CGRAM 的首地址。这可以使用“Set CGRAM Address” 命令，或者使用 LCDwrite() 函数的宏命令：

```
#define LCDsetG( a) LCDwrite( LCDCMD, (a & 0x3F) | 0x40)
```

为了生成两个 5×7 的字符图案，其中一个是飞机的头部，另一个是飞机的尾部，我们使用 putLCD() 函数。每个字节的数据将有 5 位用来定义图案的行。当每个字符的最后一行定义完毕，还需要插入一个字节（第 8 个字节数据）来对齐下一字符块。

```
// generate two new characters
LCDsetG(0);
putLCD( 0b00010);
putLCD( 0b00010);
putLCD( 0b00110);
putLCD( 0b11111);
```

```
putLCD( 0b00110);
putLCD( 0b00010);
putLCD( 0b00010);
putLCD( 0);      // alignment

putLCD( 0b00000);
putLCD( 0b00100);
putLCD( 0b01100);
putLCD( 0b11100);
putLCD( 0b00000);
putLCD( 0b00000);
putLCD( 0b00000);
putLCD( 0);      // alignment
```

现在，两个新的图案可以分别通过字符生成表的代码 0 和代码 1 来访问。

要将缓冲器指针放回到数据 RAM 缓冲器，需要使用下面的宏：

```
#define LCDsetC( a) LCDwrite( LCDCMD, (a & 0x7F) | 0x80)
```

注意，显示的第一行是对应于 DDRAM 缓冲器的地址 0 到 0xf 的，无论显示规模（每一行实际显示的字符数）的大小，第二行对应的地址都是 0x40 到 0x4f。

这里，还有一个简单的延时机制（再次使用 Timer1），对于飞机的按时飞行和可视化是很有必要的。LCD 的显示应该慢慢地，不然，若显示闪烁得太快，则像是幽灵在出没一样：

```
#define TFLY 9000        // 9000 x 16us = 144ms
#define DELAY() TMR1=0; while( TMR1<TFLY)
```

现在该设计一个简单的算法，让小飞机在主循环中飞行了。其代码如下：

```
// main loop
while( 1)
{
    // the entire plane appears at the right margin
    LCDsetC(0x40+14);
    putLCD( 0); putLCD( 1);
    DELAY();

    // fly fly fly (right to left)
    for( i=13; i>=0; i--)
    {
        LCDsetC(0x40+i);        // set the cursor to the next position
        putLCD(0); putLCD(1);   // new airplane
        putLCD(' ');            // erase the previous tail
        DELAY();
    }

    // the tip disappears off the left margin, only the tail is visible
    LCDsetC(0x40);
    putLCD( 1); putLCD(' ');
    DELAY();

    // erase the tail
    LCDsetC(0x40);              // point to the left margin of the 2nd line
    putLCD(' ');
```

```
    // and draw just the tip appearing from the right
    LCDsetC(0x40+15);         // point to the right margin of the 2nd line
     putLCD 0);
    DELAY();

} // repeat the main loop
```

尽情地和 PIC24 一起飞翔吧!

9.4 飞后小结

在本章中学习了如何使用并行主控制端口来驱动 LCD 显示模块。实际上，已经揭开更深层次的内容。同时，由于 LCD 显示模块是一个速度相对较慢的外设，因此很难看出使用 PMP 代替传统的位脉冲 I/O 控制方法的优势所在。其实，即使是访问如此简单慢速的外设时，PMP 也显示出了两个重要的优点。

- 控制信号的时序、序列和多路复用总是需要匹配特征参数，以避免总线崩溃和/或不可靠操作的危险，如编码错误和/或突发事件和时序问题（中断、错误等）。
- MCU 完全不需要考虑外部总线，允许所有更高优先级的任务同时执行。

9.5 给 C 语言专家的提示

正如前面介绍过的，当使用异步串行接口时，可以通过替换“stdio.h”库中的低级 I/O 程序，特别是“write.c”，来重定向输出到 LCD 显示。对于前面的例子，可以为标准流(stdin、stdout 和 stderr）提供到 UART2 的重定向，并为实现 LCD 显示插入如下所示的第 4 个流：

```
/*
** write.c
** replaces stdio lib write function
**
*/

#include <p24fj128ga010.h>
#include <stdio.h>

#include "conU2.h"

#include "LCD.h"

int write(int handle, void *buffer, unsigned int len)
{
    int i, *p;
    const char *pf;

    switch (handle)
    {
    case 0: // stdin
    case 1: // stdout
    case 2: // stderr
           for (i = len; i; --i)
               putU2( *(char*)buffer);
```

```
            break;

        case LCD: // additional stream
            for (i = len; i; --i)
                putLCD( *(char*)buffer);
            break;
        default:
            break;
        } // switch
        return(len);
    } //write
```

作为备选方案，读者可以将“stdout”流重定向到 LCD 显示作为应用的主要输出，而将“stderr”流重定向到串行口用作调试目的。

同时，读者可能想修改 putLCD() 函数来解释诸如'\n'的特殊字符，并开始下一行，或者引入一些 ANSI 转义码，以至于就像在终端控制台上一样能够定位光标的位置和清除屏幕（使用本章定义的宏命令）。

9.6　提示与技巧

由于 LCD 显示是一个速度较慢的外设，因此像在本章中那样等待它在紧张（封闭）的循环中完成指令，是非常浪费 MCU 周期的。更好的方案是在 FIFO 缓冲器中捕获 LCD 指令，并使用中断机制来周期性地调度指令的执行。换而言之，在程序执行的背景下，应该使用中断来处理缓慢过程的多任务。

在 Explorer16 演示板中，就有一个叫作“LCD.c”的例子是使用这种机制的。

9.7　练习

(1) 增强 putLCD() 函数，使它能正确解释下面的字符。

- '\n'：换行。
- '\r'：光标回到当前行的起始位置。
- '\t'：前进到固定的表格位置。

(2) 增强 putLCD() 函数，使它能介绍下面的 ANSI 转义码：

- '\x1b[2J'：清屏。
- '\x1b[1,1H'：光标归位。
- '\x1b[n, mH'：将光标置于 n 行 m 列。

9.8　推荐书目

- Bentham，J.

  TCP/IP lean, Web Servers for Embedded Systems

  CMP Books, Lawrence, Kansas

  这本书介绍了 TCP/IP 协议（因特网的基础）是如何使用“几行”简单的 C 代码就能实现的，将读者带进一个更高级的层次。Jeremy 知道如何让事情保持“简单”，这在每一

个嵌入式控制程序中都是必要的。

9.9 网上链接

- http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=n011993
  这是Microchip应用指南第833条的网上链接,是对所有PIC微控制器提供的免费TCP/IP协议栈。
- http://www.microchip.com/stellent/Idcplg?idcService=SS_GET_PAGE&nodeId=1824&appnote=en012108
  应用说明第870条描述了Microchip TCP/IP应用的简单网络管理协议。



第 10 章　模拟的世界

本章内容

- ▶首次转换
- ▶自动采样定时
- ▶开发演示程序
- ▶开发游戏
- ▶温度测量
- ▶Breath-Alizer 游戏

毫无疑问，无论试验多少次，都不可能飞出两条完全相同的路线。着陆就是一个很好的例子。即使是最有经验的机长，也会偶尔出错。当飞机着地又“弹起”的时候，相信乘客都能注意到这点。着陆会出什么问题呢？为什么就这么难呢？

实际上，不管飞行员多么用心，影响飞机着陆的因素不是每次都完全一样的。风速和风向不停地在变化，引擎的性能也在变化，甚至是机翼也会因为温度变化而产生微小的形变，还有飞行员的反应（和警惕性）也在变化。所有这些就组成了无限种不可预知的因素，从而可能导致无限种错误。

人类生活在一个模拟的世界里。所有的输入变量、温度、风速和风向都是模拟量。人类所有感觉器官的输入都是模拟量。而输出，如飞行员控制飞机的动作，也是模拟量。随着时间的推移，人类学会了解释（或者可以说是转换）外部世界的所有模拟输入，然后作出最佳的判决。熟能生巧！

在嵌入式控制中，来自模拟世界的信息首先需要转换成数字量。模数转换模块就是“现实”世界的关键接口之一。

10.1　飞行计划

PIC24 系列是为嵌入式控制应用设计的，因此也做好了和外界模拟量打交道的准备。一个高速的模数转换器（ADC），每秒可实现 500 000 次转换，可以连接带有快速检测模拟输入的多路输入复用器和高分辨率采样的所有模型。本章，将介绍如何使用 PIC24FJ128GA010 系列的 10 位 ADC 模块，在 Explorer16 演示板上实现两个简单的测量：首先是读取电位计的电压，然后是从温度传感器上读取电压输出。

10.2　飞前备忘录

除了常用的软件工具，如 MPLAB IDE、MPLAB C30 编译器和 MPLAB SIM 仿真器，本章还需要用到 Explorer16 演示板和 MPLAB ICD 电路内调试器。

10.3 飞行

同 PIC24 内的其他外设一样，使用模数转换器的第一步，就是要熟悉它的模块构造和主要的控制寄存器。是的，这意味着要再次阅读数据表和 Explorer16 用户指南的原理图。首先来阅读 ADC 模块的方框图，如图 10-1 所示。

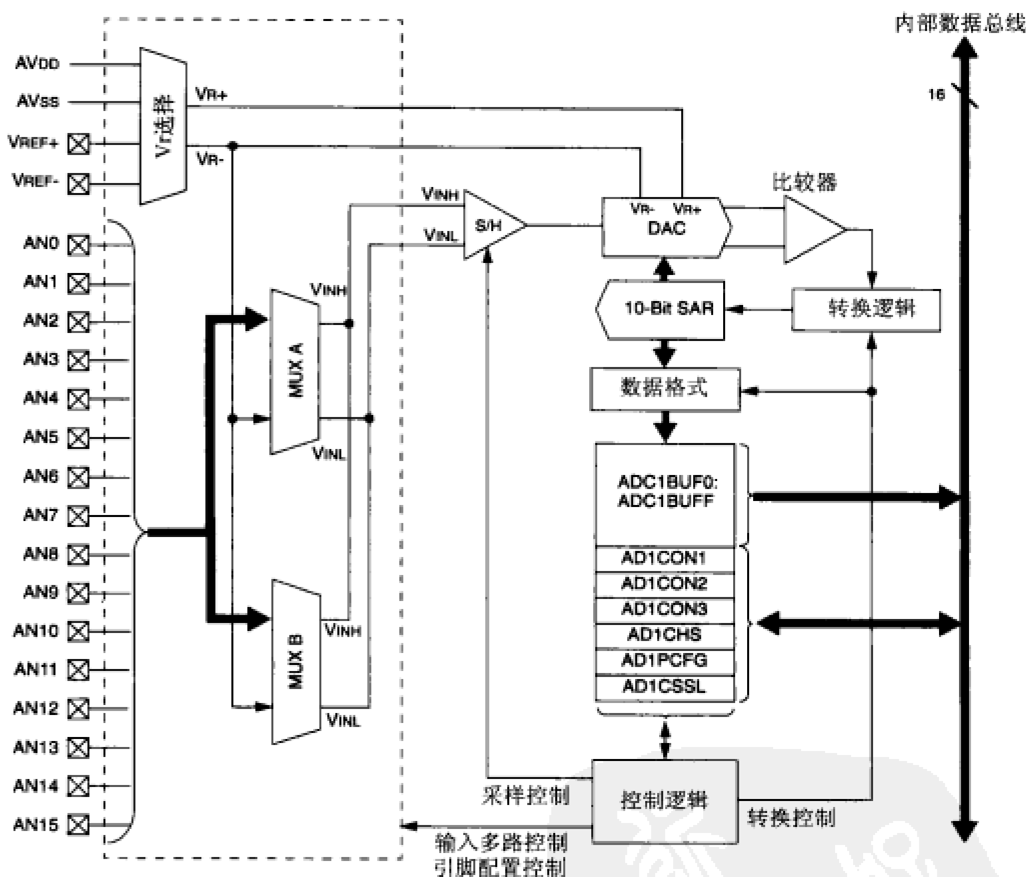


图 10-1 ADC 模块的方框图

这个看起来相当复杂的结构，具有很多有趣的特性。

- 最多 16 个输入引脚可以用于接收模拟输入。
- 两个多路输入复用器可以用于选择不同的模拟输入通道和参考电源。
- 10 位转换器的输出可以设置成整数或定点数、有符号或无符号 16 位输出。
- 控制逻辑允许多种自动的转换序列，并且与其他相关的模块和输入信号同步。
- 转换的输出保存在已配置为序列扫描或简单 FIFO 缓冲的 16 位宽、16 字长的缓冲器。

以上特性都需要正确地设置许多控制寄存器。在刚开始的时候，可能这些设置会让读者头晕眼花。因此，这里先从最简单的例子开始：从 Explorer16 演示板上 R6 电位计的位置读取数据，如图 10-2 所示。

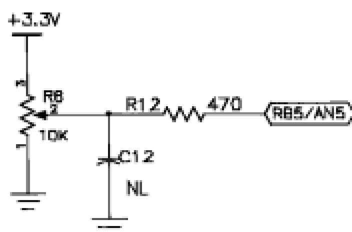


图 10-2 Explorer16 演示板的 R6 电位计

10 kΩ电位计直接连接到供电电源端，因此可以输出 0~3.3V 的电压。该电位计还连接到对应于 ADC 多路输入复用器的 AN5 模拟输入的 RB5 引脚。

在按照合适的备忘录生成新的项目后，可以生成新的源文件“pot.c”（包括常用的头文件和两个实用常量的定义）。第一个常量（POT）定义为电位计的输入通道，第二个常量（AINPUTS）作为 mask 类型，有助于用户定义哪个输入作为模拟量，哪个作为数字量。

```
/*
** It's an analog world
** Converting the analog signal from a potentiometer
*/

#include <p24fj128ga010.h>

#define POT      5        // 10k potentiometer connected to AN5 input
#define AINPUTS 0xffef   // Analog inputs for Explorer16 POT and TSENS
```

ADC 控制寄存器的初始化只需要一个简单的函数 initADC()，将产生以下既定的初始配置。

- ❑ AD1PCFG 决定模拟输入通道的 mask 类型：0 表示模拟输入，1 表示数字输入。
- ❑ AD1CON1 置位将转换设置为由采样周期完成自动触发，并将输出格式化为右对齐的简单无符号整数。
- ❑ AD1CSSL 在扫描函数无效（只有一个输入）时清零。
- ❑ AD1CON2 决定 MUXA 的用途，并且将 ADC 的参考输入连接到模拟输入引脚 AVdd 和 AVss。
- ❑ AD1CON3 选择转换时钟源和分频器。
- ❑ 设置 ADON，让整个 ADC 外设准备就绪。

```
void initADC( int amask)
{
    AD1PCFG = amask;            // select analog input pins
    AD1CON1 = 0;                // manual conversion sequence control
    AD1CSSL = 0;                // no scanning required
    AD1CON2 = 0;                // use MUXA, AVss and AVdd are used as Vref+/-
    AD1CON3 = 0x1F02;           // Tad = 2 x Tcy = 125ns >75ns
    AD1CON1bits.ADON = 1;       // turn on the ADC
} //initADC
```

将 amask 作为参数传递给初始化程序，在后面的应用中就可以灵活地使用多路输入通道了。

10.3.1 首次转换

实际上，模数转换由两步组成。首先，需要对输入电压信号进行采样，然后断开输入，将采样电压转换成数字量。这两种不同的操作由 AD1CON1 寄存器中两个不同的位控制：SAMP 和 DONE。两种操作的定时对于测量的准确性非常关键。

- 在采样阶段，外部信号连接的内部电容需要充电到输入电压。必须给电容足够的充电时间，而这个时间与输入信号源的阻抗（在本例中小于 5 kΩ）以及内部电阻值成比例关系。一般来说，采样时间越长，结果越好，还要考虑输入信号的频率问题（在本例中可以忽略）。
- 在转换阶段，定时取决于选择的 ADC 时钟源。通常是由主 CPU 时钟信号通过分频器或者独立的 RC 振荡器生成的。为简单起见，RC 振荡器是一个不错的选择，特别当转换是在休眠（低功耗）模式下进行时，那时 CPU 时钟是关闭的。更多的情况下，晶振时钟分频器是一个更常见的选择，因为它可以提供与 CPU 同步的操作，对于内部噪声有更好的抗干扰能力。转换时钟应该使用尽可能最快的，并且还要和 ADC 模块的指标兼容（在本例中，Tad 需要大于 75 ns，也就是最小时钟的一半）。

下面是基本的转换程序：

```
int readADC( int ch)
{
    AD1CHS  = ch;                 // 1. select analog input channel

    AD1CON1bits.SAMP = 1;         // 2. start sampling

    TMR1 = 0;                     // 3. wait for sampling time
    while (TMR1< 100);            // 6.25 us

    AD1CON1bits.DONE = 1;         // 4. start the conversion

    while (!AD1CON1bits.DONE);    // 5. wait for the conversion to complete

    return ADC1BUF0;              // 6. read the conversion result

} // readADC
```

10.3.2 自动采样定时

正如读者所见，在使用上面的基本程序时，要为采样提供准确的定时，即需要为采样任务分配一个定时器以及执行两个等待循环。不过 PIC24 有一个新增的功能，可以提供更多的自动处理。采样操作是可以自定时的，加入一个很小的输入源阻抗就足以提供 32×Tad 的最大采样时间（在本例中是 32×120 ns=3.8 μs）。将 AD1CON1 寄存器的 SSRC 位设置成 0b111，那么在自定时采样周期结束后，系统就会自动开始转换。采样周期本身是通过 AD1CON3 寄存器的 SAM 位设置的。下面的改进代码示例使用了自定时采样和转换触发功能：

```
void initADC( int amask)
{
    AD1PCFG = amask;              // select analog input pins
    AD1CON1 = 0x00E0;             // automatic conversion start after sampling
```

```
    AD1CSSL = 0;              // no scanning required
    AD1CON2 = 0;              // use MUXA, AVss and AVdd are used as Vref+/-
    AD1CON3 = 0x1F02;         // Tsamp = 32 x Tad; Tad=125ns
    AD1CON1bits.ADON = 1;     // turn on the ADC
} //initADC
```

注意，使用自定时采样来自动触发转换操作有以下两个优点：

- 正确的采样定时，不需要用户使用延时循环和/或其他资源；
- 一条命令（从采样开始）就可以完成采样和转换序列。

当 ADC 设置好了以后，转换和输出读取就很简单了，其步骤如下。

- AD1CHS 用于为 MUXA 选择输入通道。
- 在 AD1CON1 寄存器中将 SAMP 置位，启动定时采样，紧跟在转换之后。
- AD1CON1 寄存器的 DONE 位在整个序列完成后马上会变成 1，说明结果已经就绪。
- 读取 ADC1BUF0 寄存器的内容，就会得到期望的转换结果。

```
int readADC( int ch)
{
    AD1CHS  = ch;                // 1. select analog input channel

    AD1CON1bits.SAMP = 1;        // 2. start sampling

    while (!AD1CON1bits.DONE);   // 3. wait for the conversion to complete

    return ADC1BUF0;             // 4. read the conversion result

} // readADC
```

10.3.3 开发演示程序

现在要做的，就是找出一个有趣的方式来把转换结果输出到 Explorer16 演示板上。很容易就可以想到把 LED 显示器连接到 PORTA 上，不过除了简单的二进制输出或者显示 10 位结果的 8 个最高有效位之外，为什么不把难度提高一点点，输出一个更能反映模拟输入的可视化结果呢？可以每次打开一个 LED，作为机械拨号盘的一个编号。下面就是用于测试模数转换功能的主程序：

```
main ()
{
    int a;

    // initializations
    initADC( AINPUTS);  // initialize the ADC for the Explorer16 analog inputs
    TRISA = 0xff00;     // select the PORTA pins as outputs to drive the LEDs

    // main loop
    while( 1)
    {
        a = readADC( POT);  // select the POT input and convert

        // reduce the 10-bit result to a 3 bit value (0..7)
        // (divide by 128 or shift right 7 times
```

```
        a >>= 7;

        // turn on only the corresponding LED
        // 0 -> leftmost LED.... 7-> rightmost LED
        PORTA = (0x80 >> a);

    } // main loop
} // main
```

在调用初始化子程序（这里提供一个将第 5 位定义成模拟输入的 mask 变量）之后，初始化 TRISA 寄存器，将引脚连接到 LED 数字输出条。然后，在主循环中，对 AN5 执行转换，将输出格式化以满足特定的显示要求。正如所配置的，10 位的转换结果将会变成一个右对齐的整数，其范围在 0 到 1 024 之间。再除以 128（或者是右移 7 次），那么范围就变成了 0 到 7。不过最终的输出还需要多增加一步转换来生成所需的 8 段 LED 显示。注意，LED 对应的 MSB 在左边，要保持电位计的顺时针移动和 LED 编号的向右移动，这需要从 0b10000000 模式开始，然后按要求右移。

构建项目（根据常用的 ICD2 调试备忘录）对 Explorer16 演示板编程。如果一切顺利，读者可以看到，当电位器从一边移向另一边时，LED 的显示也相应地左右移动。

10.3.4 开发游戏

好了，上面的例子并没有想像中那么有趣。毕竟，读者使用了 16 MIPS 的 16 位机来执行每秒 200 000 次的模数转换（32Tad 的采样+12Tad 的转换，其中 Tad=125 ns，读者自己会算了吧），仅仅是得到了 3 位的结果，看见了 LED 显示器闪亮。不如来开始一个更有挑战性的项目吧。现在来开发一个一维的 “Whac-A-Mole（打地鼠）”小游戏吧！

打开另一个由 PIC24 控制的 LED（地鼠），它可能会稍暗一点，以区别于玩家控制的 LED（锤子）。移动锤子（较亮的 LED），转动电位计，直至到达地鼠（较暗的 LED），然后“用力打”！这时，另一只新的地鼠会出现在注意的位置上，游戏继续。

使用伪随机数发生器函数 rand()（在“stdlib.h”文件中定义）在这里大派用场，因为所有的（计算机）游戏都需要一定程度的不可知。在这里使用它来确定新的地鼠从哪里冒出来。

将第一个项目的源文件保存在“LEDgame.c”中，然后生成一个全新的项目。修改 main() 函数，加入一些新的代码：

```
main ()
{
    int a, r, c;

    // 1. initializations
    initADC( AINPUTS);  // initialize the ADC for the Explorer16 analog inputs
    TRISA = 0xff00;     // select the PORTA pins as outputs to drive the LEDs

    // 2. use the first reading to randomize the number generator
    srand( readADC( POT));
    r = 0x80;
    c = 0;
```

```
    // 3. main loop
    while( 1)
    {
        a = readADC( POT);  // select the POT input and convert

        // 3.1  reduce the 10-bit result to a 3 bit value (0..7)
        // (divide by 128 or shift right 7 times
        a >>= 7;

        // 3.2 turn on only the corresponding LED
        // 0 -> leftmost LED.... 7-> rigtmost LED
        a = (0x80 >> a);

        // 3.3 as soon as the cursor hits the random dot, generate a new one
        while (a == r )
            r = 0x80 >> (rand() & 0x7);

        // 3.4 display the user (bright) LED and food (dim) LED
        if ((c & 0xf) == 0)
            PORTA = a + r;   // add food LED only 1/16 of the times (dim)
        else
            PORTA = a;       // always display the user LED (bright)

        // 3.5 loop counter
        c++;

    } // main loop
} // main
```

- 在程序段 1 中，执行模数转换模块的惯常初始化，将 PORTA I/O 引脚连接到 LED 条。
- 在程序段 2 中，首次读取电位计的值，并且利用该值作为随机数生成器的引子。这样可以使每次的游戏都不同，只要电位计不总是位于最左边或者最右边。否则引子的值不是 0 就是 1 023，每次游戏重启的时候就会重复产生相同的伪随机序列。
- 在程序段 3 中，主循环开始，类似于前一个例子，读取 10 位整数，然后减少到 3 个最高有效位。(如程序段 3.1 所示。)
- 在程序段 3.2 中，和前面一样，执行 LED 显示器位置 “a” 的转换操作。不过在程序段 3.3 中，情况就有趣多了。如果 “a” 表示的玩家 LED 位置和 “r” 表示的地鼠 LED 位置重合，就会马上计算出新的随机位置。这个操作需要不停地执行 “while” 循环，因为每次新随机量 “r” 的计算，都有可能生成相同的数（准确地说是 1/8 的概率，如果伪随机数发生器正常工作的话）。换言之，新出现的 “地鼠” 可能正好在 “锤子” 的正下面。那样就毫无挑战性和体育精神了，是不是？
- 程序段 3.4 和程序段 3.5 是用于显示和区分两个 LED 的。要在显示条上显示两个 LED，程序员只需要简单地 “加上” 两个二进制代码 “a” 和 “r”，但是玩家就很难区分谁是谁了。要将 “地鼠” LED 设成暗一点，需要在主循环中交替执行两个周期，一个周期用于两个 LED 都亮的情形，另一个周期用于只有 “锤子” LED 亮的情形。因为每一秒中，主循环都会执行成百上千次，所以，肉眼就会觉得 “地鼠” LED 比较暗，因为在

一部分的周期里，它是不发光的。例如，如果将“地鼠”LED设成每16个周期亮一次，那么它的亮度就只有“锤子”LED的1/16。

- 程序段3.5中的计数器“c”一直增加将有助于实现这种机制。
- 在程序段3.4，只关注计数器的4个最低有效位（0~15），只有当它为0b0000时，才让“地鼠”LED发光。而在其余的15个循环中，只有“锤子”LED发光。

创建项目，并下载到Explorer16演示板。读者必须承认，现在的游戏有趣多了！

10.3.5 温度测量

下面继续训练课程。在Explorer16演示板上有一个温度传感器，实际上，它是一个具有很好的线性电压输出特性的Microchip TC1047A集成式温度传感器件。该器件是SOT-23（3个引脚，表面贴装）封装，因此体积非常小。它的功耗只有35 μA（典型值），而电源可以是2.5 V到5.5 V。输出电压与电源电压无关，是温度（特别是0.5 °C之内）的绝对线性函数，斜率为10 mV/°C。根据图10-3中的方程可知，调整电压的偏移量可找出绝对的温度值。

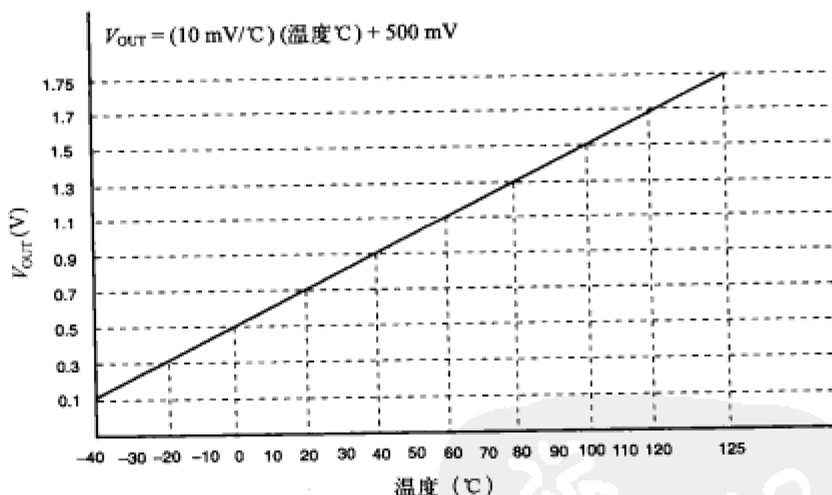


图10-3 TC1047输出的电压-温度特性曲线

现在再次使用PIC24的模数转换器，把输出电压转换成数字信息。按照Explorer16演示板的结构图，将温度传感器连接到AN4模拟输入通道，如图10-4所示。

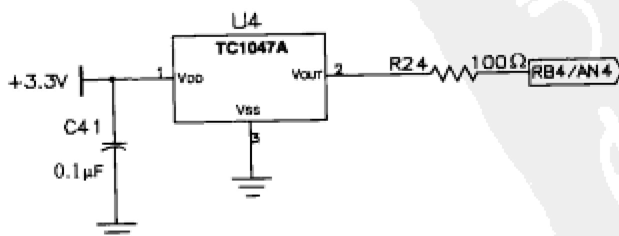


图10-4 Explorer16演示板上TC1047A温度传感器的连接

可以利用前一个练习中构造的 ADC 函数，将它们添加到新的项目“TSense”中，并将原来的源文件保存为“Tsense.c”。

现在开始修改代码，包括一个新常量的定义：TSENS，用于将 ADC 输入通道分配给温度传感器：

```
/*
** It's an analog world
** Converting the analog signal from a TC1047 Temperature Sensor
*/

#include <p24fj128ga010.h>

#define POT      5        // 10k potentiometer connected to AN5 input
#define TSENS    4        // TC1047 Temperature sensor with voltage output
#define AINPUTS 0xffcf    // Analog inputs for Explorer16 POT and TSENS

// initialize the ADC for single conversion, select Analog input pins
void initADC( int amask)
{
    AD1PCFG = amask;        // select analog input pins
    AD1CON1 = 0x00E0;       // auto convert after end of sampling
    AD1CSSL = 0;            // no scanning required
    AD1CON3 = 0x1F02;       // max sample time = 31Tad, Tad = 2 x Tcy = 125ns >75ns
    AD1CON2 = 0;            // use MUXA, AVss and AVdd are used as Vref+/-
    AD1CON1bits.ADON = 1; // turn on the ADC
} //initADC

int readADC( int ch)
{
    AD1CHS  = ch;                  // select analog input channel
    AD1CON1bits.SAMP = 1;          // start sampling, auto-conversion will follow
    while (!AD1CON1bits.DONE);     // wait to complete the conversion
    return ADC1BUF0;               // read the conversion result
} // readADC
```

正如读者所见，不需要对 ADC 的特性或者转换序列作任何的改动。不过要在 LED 条上显示结果可能会遇到麻烦。温度传感器有一定的噪声，因此要获得稳定的读数，通常需要滤波。对每 16 个采样值求平均值，可以得到比较清晰的结果：

```
a = 0;
for ( j= 16; j >0; j--)
     a += readADC( TSENS);  // add up 16 successive temperature readings
i = a >> 4;                 // divide the result by 16 to obtain the average
```

可是只使用 LED 条如何显示结果呢？

可以把转换结果的最高有效位以二进制或者 BCD 的形式在 LED 上显示出来，不过那样就不好玩了。不如对温度的显示也使用相似的（单 LED）指示移动的方法吧。

在主循环之前，完成初始温度值的采样，然后使用它作为相对于中心显示条位置的偏移量。在主循环中，可以更改点的位置，温度升高，则点向右移；温度降低，则点向左移。下面就是全新的温度传感器例子的完整代码：

```
main ()
{
    int a, i, j;

    // 1. initializations
    initADC( AINPUTS);  // initialize the ADC for the Explorer16 analog inputs
    TRISA = 0xff00;     // select the PORTA pins as outputs to drive the LEDs
    T1CON =   0x8030;   // TMR1 on, prescale 1:256 Tclk/2

    // 2. sample initial temp value
    a = 0;
    for ( j= 16; j >0; j--)
         a += readADC( TSENS);  // read the temperature
    i = a >> 4;
    // this will give the central bar reference

    // 3. main loop
    while( 1)
    {
        // 3.1 read a new (averaged) temperature value
        a = 0;
        for ( j= 16; j >0; j--)
        {
              TMR1 = 0;
              while ( TMR1 < 3900);  // 3900 x 256 x Tcy ~= 1sec
              a += readADC( TSENS);  // read the temperature
        }
        a >>= 4;                     // averaged over 16 readings

        // 3.2 compare with the initial reading and move the bar 1 pos. per C
        a = 3 + (a - i);

        // 3.3 keep the result in the value range 0..7, keep the bar visible
        if ( a > 7)
            a = 7;
        if ( a < 0)
            a = 0;

        // 3.4 turn on the corresponding LED
        PORTA = ( 0x80 >> a);

    } // main loop
} // main
```

- 在程序段 3.2 中，求取初始读数“i”和新的均值读数“a”的差值。所得结果显示在中心位置，当两者相差为 0 时，正中间的 LED 就会点亮。
- 在程序段 3.3 中，检查结果是否超出显示范围。一旦两者的差小于-3，就只显示最左边的 LED 显示器。当差大于+4 时，就显示最右边的 LED 显示器。
- 在程序段 3.4 中，采用前一个例子中的方法显示结果。

为了让这个练习带给读者更多视觉上的快感，建议读者再加上一个延时循环（为方便起见，就加在程序段 3.1 中的循环前面）。它会降低运行的速度，把显示的刷新速度减慢（最终是整个主循环周期）到大约每秒 1 次。当温度的读数太接近中间值时， LED 太快的刷新只会让人眼

花缭乱。

按照常用的备忘录构造项目，并且下载到 Explorer16 演示板上。

在找出演示板上的温度传感器后（提示：它位于 PIC24 处理器模块的左下角，看起来像是一个表面贴装晶体管），运行程序，可以通过触摸或者吹热/冷空气，观察温度的细微变化，并在附近移动光标。

10.3.6　Breath-Alizer 游戏

利用温度传感器，可以把前面两个练习融合在一起，组合成一个更好玩的游戏。暂且把它叫做“Breath-Alizer”游戏，也就是利用温度传感器控制“锤子”来打“地鼠”的游戏。向传感器吹热空气，锤子向右；吹冷空气，锤子就向左。尽情玩吧！

```
main ()
{
    int a, i, j, k, r;

    // 1. initializations
    initADC( AINPUTS);  // initialize the ADC for the Explorer16 analog inputs
    TRISA = 0xff00;     // select the PORTA pins as outputs to drive the LEDs
    T1CON = 0x8030;     // TMR1 on, prescale 1:256 Tclk/2

    // 2. use the first reading to randomize the number generator
    srand( readADC( TSENS));
    // generate the first random position
    r = 0x80 >> (rand() & 0x7);
    k = 0;

    // 3. compute the average value for the initial reference
    a = 0;
    for ( j= 16; j >0; j--)
         a += readADC( TSENS);  // read the temperature
    i = a >> 4;

    // 5. main loop
    while( 1)
    {
        // 5.1 take the average value over 1 second
        a = 0;
        for ( j= 16; j >0; j--)
        {
            TMR1 = 0;
            while ( TMR1 < 3900)  // 16 x 3900 x 256 x Tcy ~= 1sec
            { // display the user LED and dim random LED
                if ((TMR1 & 0xf) == 0)
                    PORTA = k + r;
                else
                    PORTA = k ;
            }

            a += readADC( TSENS);  // read the temperature
        }
        a >>= 4;                   // averaged over 16 readings
```

```
        // 5.2 compare with the initial reading and move the bar 1 pos. per C
        a = 3 + (a - i);
        // keep the result in the value range 0..7, keep the bar visible
        if ( a > 7)
            a = 7;
        if ( a < 0)
            a = 0;
        // update the user LED
        k = ( 0x80 >> a);

        // 5.3 as soon as the user hits the random LED, generate a new position
        while (k == r )
            r = 0x80 >> (rand() & 0x7);

    } // main loop
} // main
```

10.4 飞后小结

本章初次介绍了 PIC24 的模数转换模块的结构和功能。在本章中，只用到了它的一个简单的功能和几个高级特性。利用 Explorer16 演示板，对两种模拟输入的模数转换性能进行了测试，希望读者能在本章的学习中获得一些乐趣。

10.5 给 C 语言专家的提示

即使 PIC24 能够快速地处理除法运算，但是也没有理由去浪费处理器周期。在嵌入式控制中，“每个”处理器周期都是精确的。如果除数是 2 的幂，整数的除法正好可以通过右移正确的数位来处理，这样的运算损耗比执行常规的除法要小得多。如果除数不是 2 的幂，那么也应该考虑转换一下。在本章的最后一个例子中，完全可以取 10 次，或者 15 次，甚至 20 次的温度采样的均值，但是最终使用的是 16，因为这样可以通过简单 4 位的右移来实现（只需一个 PIC24 指令周期）除法运算。

10.6 提示与技巧

如果需要的采样时间大于最大可用时间（32 × Tad），那么可以考虑先延长 Tad，或者更好的方法是将事情倒过来，（在转换结束后）启动自动采样。这样无论转换是否发生，采样电路总是在充电。手动清零 SAMP 位可以触发实际的转换开始。

此外，可以使用 Timer3 周期性地对 SAMP 控制位清零（即 AD1CON1 中的 SSRC 位），允许在 ADC 转换结束时发生中断，可以为获得最小的 MCU 负荷提供最宽范围的采样周期。无需等待循环，仅需一个周期性的中断，就可以准备就绪并读取转换结果。

10.7 练习

使用 ADC FIFO 缓冲器收集转换结果；设置 Timer3 用于自动转换和中断机制，只在缓冲器满的时候调用函数，而且温度值已经是平均值。

10.8 推荐书目

- Baker, B.
  A Baker's Dozen: Real Analog Solutions for Digital Designers
  Newnes, Burlington, MA
  没有哪本书比它对模数转换器介绍得更详细了。

10.9 网上链接

- http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2102&param=en021419&pageId=79&pageId=79
  温度传感器有很多的用途和接口选择，包括直接 I^2C 或者 SPI 数字输出。



第三部分

跨 国 飞 行

恭喜！读者又完成了一个阶段的课程，具备了执行更复杂的飞行任务的能力。下面将进入第三个阶段的训练，也就是最后一组训练，即读者将进行跨国飞行的训练。不仅是简单的机场周围的标识，也不仅仅是起飞和降落或者训练场的机动限制——读者终于可以飞出去了。

这一部分将开发几个控制外部模块的新项目。由于例子变得更复杂，读者不仅需要真实的演示板（Explorer16），还需要改进和利用原型区来增添新的功能。下面的章节将提供一些简单的原理图和器件型号。网站“FlyingthePIC24.com”（和/或“ProgrammingthePIC24.com”）上有更多的扩展板和原型模型可供选择，这有助于读者更好地体验高级项目带来的乐趣。



第 11 章 输入捕捉

本章内容

- PS/2 通信协议
- PIC24 连接 PS/2
- 输入捕捉
- 使用激励脚本测试输入捕捉
- 仿真
- 测试 PS/2 接收子程序
- 仿真器规范
- 另一种方法——变化通知
- 开销计算
- 第三种方法——I/O 查询
- 测试 I/O 查询方法
- 方案的性价比
- 完成接口：添加 FIFO 缓冲器
- 完成接口：解码按键码

正如在前面章节介绍过的，除了最小的飞机以外，先进的电子设备正在快速地进入驾驶舱。正当玻璃（LCD）显示器取代旧式的蒸汽仪表时，GPS 卫星接收器和其他的仪器正在彩色地图上实时地描绘飞机位置，包括地形高度以及可精确到分钟的卫星气象信息。飞行员能够进入导航系统的整个飞行计划，就像电视游戏一样，可以沿着移动地图上的路径来飞行。不过，这些仪器之间的相互干扰又是一个巨大挑战。由于计算机的应用，每个仪器都由不同的菜单系统以及一大堆的操纵杆和按钮来控制，允许飞行员快速、直观地输入信息。然而，目前驾驶舱的有限空间严重地限制了输入设备的种类和数量，其中最重要的部分——至少对于第一代来说——已经被 VHF 无线装置的操纵杆和按钮所取代。

如果读者的汽车上装备有 GPS 导航系统，并且尝试过在陌生城市的高速公路上通过旋转小小的操纵杆来查询街道地址的详细信息，那么读者应该知道这有多难吧。键盘是很多高级航空（航空电子）系统输入的逻辑层接口。在商用喷气机的驾驶舱内，键盘已经是很常见的设备，并且正在逐渐进入更小型的通用飞机。那么在以后的汽车里会出现键盘吗？

11.1 飞行计划

随着 USB 总线的出现，计算机终于可以摆脱一系列从第一台 IBM 个人电脑就开始沿用了几十年的“传统”接口了。PS/2 鼠标和键盘接口就是其中之一，这个变化导致很多“老式”键盘在市场上严重滞销，哪怕是新型的 PS/2 键盘的售价也非常低。因此，这就为今后的 PIC24 项目的强大输入能力创造了有利的契机，同样也让更多人去研究不同输入接口方式及其优劣比较。本章将实现软件状态机，使用中断方法更新开发经验，并且学习新的外围设备。

11.2 飞行

PS/2 物理端口使用 5 针 DIN 或者 6 针迷你型 DIN 连接器，如图 11-1 所示。前者常见于最初的 IBM PC-XT 和 IBM PC-AT 系列，不过现在已经不用了。更小的 6 针迷你型版本在近年更

为常见。就具体的针脚而言，这两种接口在电气特性上是相同的。

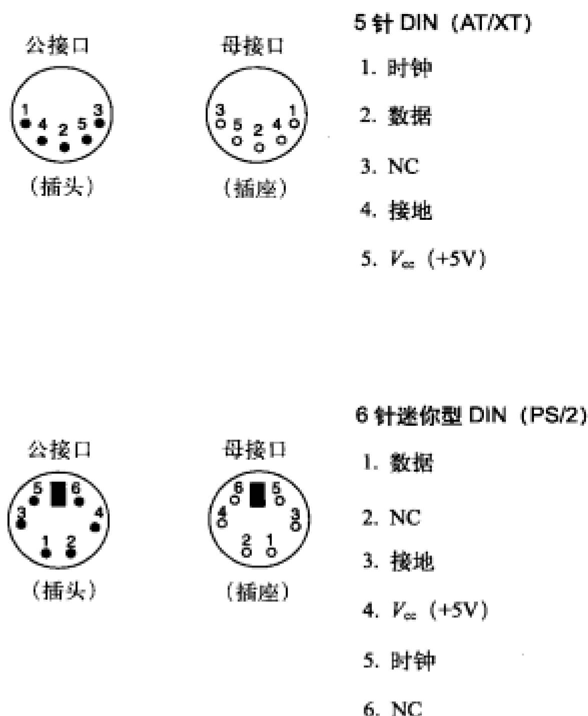


图 11-1　PS/2 物理接口

主机必须提供 5V 的电压。电流损耗随着键盘的模型和年份而变，取值范围大概在 50 mA 到 100 mA（最早的可以达到 275 mA）。

数据线和时钟线都是开集电极的，带有上拉电阻（1 kΩ ~10 kΩ），因此可以双向通信。在正常模式下，是由键盘来驱动两条线路向个人电脑发送数据。在必要时，电脑可以控制键盘的配置并且改变状态 LED（“Caps” 和 “Num Lock” 键）。

11.2.1　PS/2 通信协议

在空闲时，数据线和时钟线都被上拉为高电平（上拉电阻位于键盘内），如图 11-2 所示。在这种条件下，键盘处于使能状态，只要按键被按下就能立刻发送数据。如果主机把时钟线置为低电平并保持 100 μs 以上，那么键盘的发送信息就会中止。如果主机先将数据线置为低电平，然后再释放时钟线，那么这就会被视为发送命令的请求。

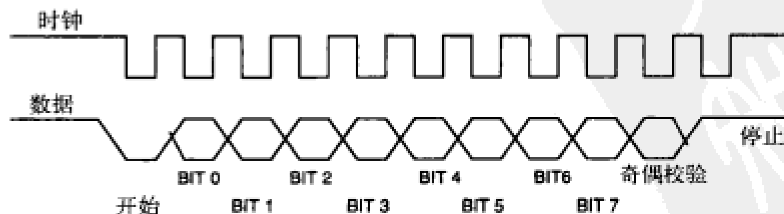


图 11-2　从键盘到主机的通信波形

该协议有趣地把前面章节介绍过的同步和异步通信协议融合在一起。它是同步的，因为提供了时钟线；但它又类似于异步协议，因为开始、停止和奇偶校验位都是用来对 8 位数据打包的。不幸的是，波特率并没有标准的定义，它可以因不同的时间、温度还有月亮的位相而变化。实际上，典型的波特率范围是 10 Kb/s 到 16 Kb/s。数据在时钟为高电平时改变。当时钟线为低电平时，数据有效。键盘总是不停地产生时钟信号，不管数据流是从键盘流向主机还是从主机流向键盘。

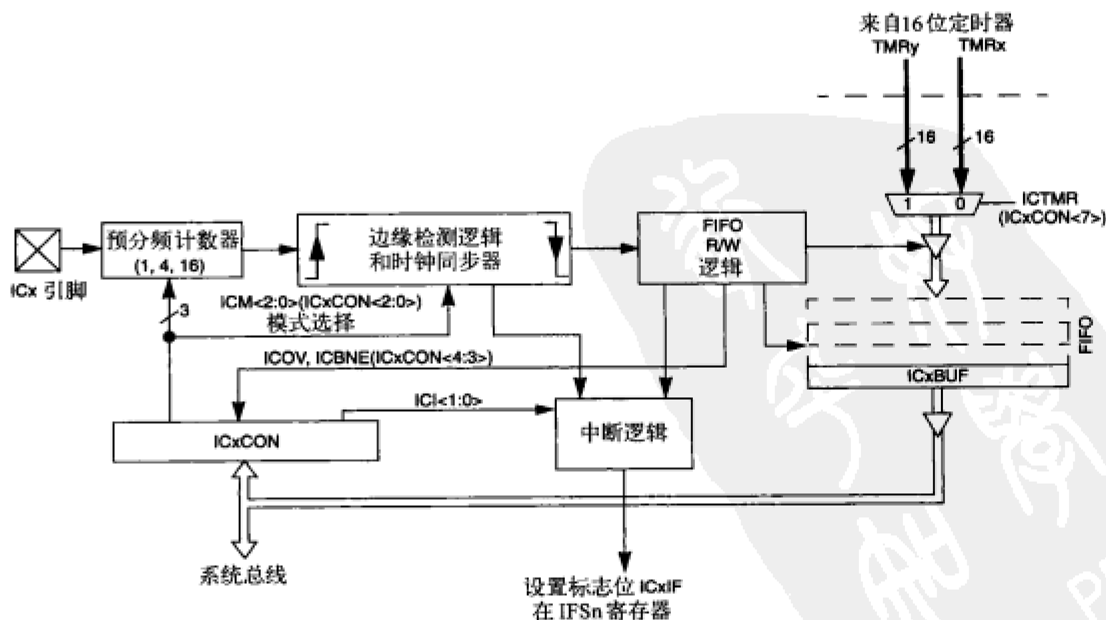
> **注解** USB 总线扮演的角色，是让每个外设成为主机的同步从机。这个规定简化了许多非实时、无优先权的多任务操作系统，如 Windows 操作系统。串行口和并行口都是异步接口，（可能都是由于这个原因导致）它们因 USB 总线的引入而演变为传统技术。

11.2.2 PIC24 连接 PS/2

PS/2 键盘的协议有一个特殊的性质，是 PIC24SPI 和 UART 接口都没有的。实际上，SPI 接口并不允许使用 11 位字（只能使用 8 位字或 16 位字），而 PIC24UART 需要周期性地发送特殊的间歇字符，来保证自动波特率检测。还要注意，PS/2 协议是基于 5V 信号的。这需要小心地选择 PIC24 的连接引脚。实际上，只能使用可以承受 5V 数字输入的引脚，这其中不包括与模数转换器复用的 I/O 引脚。

11.2.3 输入捕捉

首先，使用输入捕捉机制，在软件中实现 PS/2 串行接口外设，如图 11-3 所示。



注：符号中“x”，是寄存器或者位名，用来表示捕捉通道的序号。

图 11-3 输入捕捉模块示意图

PIC24FJ128GA010 有 5 个输入捕捉模块，分别对应连接到与 PORTD 引脚 8、9、10、11 和 12 复用的 IC1~IC5 引脚。

每个输入捕捉模块都由唯一对应的控制寄存器 ICxCON 控制，并且和 Timer2 或者 Timer3 一起工作。

下面的任一事件都可能触发输入捕捉：

- ❑ 上升沿；
- ❑ 下降沿；
- ❑ 上升和下降沿；
- ❑ 第四上升沿；
- ❑ 第十六上升沿。

所选定时器的当前值保存在 FIFO 缓冲器中，可以通过对应的 ICxBUF 寄存器读取。除了捕捉事件外，在指定数量的事件（每次、每两次、每三次或每四次）后还可以产生中断。

为了使用输入捕捉外设并从 PS/2 键盘接收数据流，可以把 IC1 输入连接到时钟线，然后设置外设在时钟的每个下降沿产生一个中断，如图 11-4 所示。

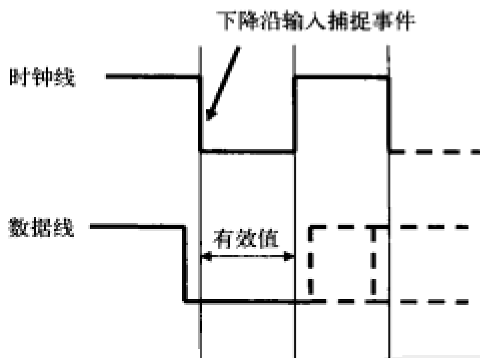


图 11-4　PS/2 接口位定时和输入捕捉触发事件

在建立新项目后，根据常用模板，键入以下初始化代码：

```
#define PS2DATA   _RG12          // any available 5V tolerant input
#define PS2CLOCK _RD8           // use the IC1 module input pin

void initKBD( void)
{
    // clear the flag
    KBDReady = 0;

    _TRISD8 = 1;                // make IC1 = RD8 pin an input (clock)
    _TRISG12 = 1;               // make the RG12 pin an input (data)
    IC1CON = 0x0002;            // use TMR3, int every capture, falling edge
    _IC1IF = 0;                 // clear the interrupt flag
    _IC1IE = 1;                 // enable the IC1 interrupt

} // void initKBD
```

同时，还要为 IC1 中断向量生成中断服务子程序。该子程序可用作状态机并按顺序执行以下的步骤，如图 11-5 所示。

(1) 验证开始位（数据线为低电平）。

(2) 输入 8 位数据，计算奇偶校验值。

(3) 验证有效的奇偶校验位。

(4) 验证停止位（数据线为高电平）。

如果以上任何一项验证失败，那么状态机会重置并且恢复到开始状态。如果接收到有效数据字节，那么就会存储到缓冲器——可以把它想像成一个邮箱——然后标志位会变为高电平，因此主程序或者其他的"消费者"子程序就会知道有效按键码已经被接收并且可以转存。要获取按键码，可以先从邮箱中复制，然后将标志位清零。

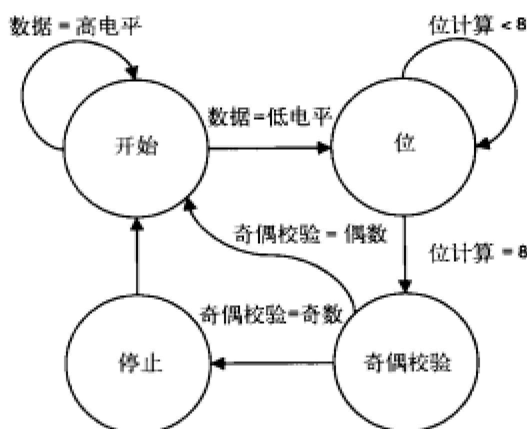


图 11-5　PS/2 接收状态机示意图

状态机只需要四个状态和一个计数器，所有的转换如表 11-1 所示。

表 11-1　PS/2 接收状态机转换表

状　态	条　件	结　果
开始	数据=低电平	初始化位计算，初始化奇偶校验，转到位状态
位	位计算<8	移位按键码，LSB 先（向右），更新奇偶校验，位计算加 1
	位计算=8	转到奇偶校验状态
奇偶校验	奇偶校验=偶数	错误。回到开始状态
	奇偶校验=奇数	转到停止状态
停止	数据=低电平	错误。回到开始状态
	数据=高电平	在缓冲器中保存按键码，设置标志位，转到开始状态

理论上需要考虑 11 个状态机，每次位状态输入不同的位计算值都当作不同的状态。但是四

状态的模式对于 C 语言实现最有效率的。下面就来定义一些维持状态机的常量和变量：

```
// definition of the keyboard PS/2 state machine
#define PS2START    0
#define PS2BIT      1
#define PS2PARITY   2
#define PS2STOP     3

// PS2 KBD state machine and buffer
int PS2State;
unsigned char KBDBuf;          // temporary buffer
int KCount, KParity,;          // bitcount and parity

// key code flag and mailbox
volatile int KBDReady;
volatile unsigned char KBDCode;
```

注解 关键字 volatile 用作声明变量的变址，告诉编译器变量的内容可以根据中断或者其他硬件机制随意地更改。在这个例子中，使用它来阻止编译器在使用这两个变量的时候采取任何的优化技术（如循环提取、程序提取等）。诚然，在这里可以忽略例子中所有的细节问题（毕竟，所有的优化都应避免出现在调试中），只需要找出当这些代码用于更复杂的项目时可能出现的最严重问题，并且尽力克服以获得最优性能。KBDReady 和 KBDcode 是两个唯一同时出现在中断服务子程序和主接口代码中的变量。

输入捕捉 IC1 模块的中断服务子程序最终可以使用简单的转换语句（执行完整的状态机）来实现。

```
void _ISR _IC1Interrupt( void)
{ // input capture interrupt service routine

    switch( PS2State){
        default:
        case PS2START:
            if ( ! PS2DAT)
            {
                KCount = 8;         // init bit counter
                KParity = 0;        // init parity check
                PS2State = PS2BIT;
            }
            break;

        case PS2BIT:
            KBDBuf >>=1;                // shift in data bit
            if ( PS2DAT)
                KBDBuf += 0x80;
            KParity ^= KBDBuf;          // update parity
            if ( --KCount == 0)         // if all bit read, move on
                PS2State = PS2PARITY;
```

```
            break;

        case PS2PARITY:
            if ( PS2DAT)
                KParity ^= 0x80;
            if ( KParity & 0x80)        // if parity is odd, continue
                PS2State = PS2STOP;
            else
                PS2State = PS2START;
            break;

        case PS2STOP:
            if ( PS2DAT)                // verify stop bit
            {
                KBDCode = KBDBuf;       // save the key code in mail box
                KBDReady = 1;           // set flag, key code available
            }
            PS2State = PS2START;
            break;

    } // switch state machine

    // clear interrupt flag
    _IC1IF = 0;

} // IC1 Interrupt
```

11.2.4 使用激励脚本测试输入捕捉方法

多孔原型区可用于 PS/2 迷你型 DIN 连接器和 Explorer16 演示板的连接，只需要为连接器扩展开发自定义的子板（PICTail）。在决定设计这种电路板前，需要确保选择的引脚和代码是可用的。在这里，MPLAB SIM 软件仿真器再次成为有用的工具。

在前面的章节中，使用了软件仿真器的 Watch 窗口，Stopwatch 秒表和逻辑分析器来验证程序是否产生正确的定时和输出，而在这里，同样需要对输入进行仿真。目前，MPLAB SIM 提供了相当多的选择和资源，这使得系统看起来让人吃惊。首先，仿真器提供两种类型的输入激励：一种是异步输入，通常由用户手动触发，另一种是同步输入，由仿真器经过特定的时间（用处理器周期或秒表示）后自动触发。脚本文件（.SCL）描述了同步激励（它可以是相当复杂的），可以通过方便的工具 SCL 发生器来生成。读者可以从调试器菜单中选择“SCL Generator→New Workbook”来调用 SCL 发生器。要得到最简单的激励脚本，用户不仅可以在指定位置按时间向特定输入引脚（以及整个寄存器）分配数值，也可以选择发生器窗口的第一栏“Pin/Register Actions”，如图 11-6 所示。

当选好计量单位后（在这个例子中是“微秒”），单击对话框窗口中最显眼的表格的第一行（“Click here to Add Signals”）。这样，用户就可以向表格添加项目了。把要仿真输入的引脚都加进去。在这个例子中，这些引脚是 PS2 数据线的 RG12 和输入捕捉引脚的 IC1，它们连接到 PS2 时钟线。现在，就可以在表中输入激励定时了。为了仿真一个普通的 PS/2 键盘传输，需要产生一个 11 周期的 10 kHz 时钟信号，如图 11-4 所示的 PS/2 键盘波形。这需要每隔 50 μs 就向定时表插入一个事件。作为示例，表 11-2 列出了添加到 SCL 发生器定时表中的触发事件，以仿真

按键码 0x79 的传送。

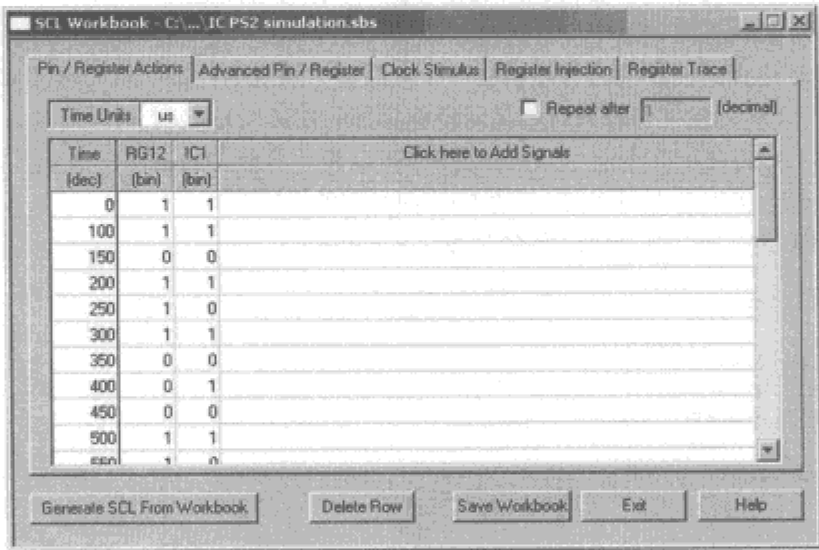


图 11-6 SCL 发生器窗口

表 11-2 用于基本的 PS/2 仿真的 SCL 发生器定时示例

时刻（us）	RG12	IC1	描 述
0	1	1	空闲状态，两条线路都为高
100	1	1	
150	0	0	第一个下降沿，开始位（0）
200	1	1	
250	1	0	位 0，按键码 LSB（1）
300	0	1	
350	0	0	位 1（0）
400	0	1	
450	0	0	位 2（0）
500	1	1	
550	1	0	位 3（1）
600	1	1	
650	1	0	位 4（1）
700	1	1	
750	1	0	位 5（1）
800	1	1	
850	1	0	位 6（1）
900	0	1	
950	0	0	位 7，按键码 MSB（0）
1000	0	1	
1050	0	0	奇偶校验位（0）
1100	1	1	
1150	1	0	停止位（1）
1200	1	1	空闲

一旦填好了定时表，读者就可以使用“Save Workbook”按钮将当前内容保存为以后使用。生成的文件是 ASCII 文件，扩展名是.SBS。理论上，读者可以通过 MPLAB IDE 编辑器或者其他基本的 ASCII 编辑器来对这个文件进行修改的，不过本书不提倡这样做。格式比美观更重要，读者可能会破坏格式的。如果读者不明白一个看似简单的表为什么会被叫做“WorkBook”，那么请打开 SCL 发生器的其他窗口（单击对话框顶部的任务栏）。读者可以看到，本例中使用的只是众多激励产生方法的其中之一，只体现来 SCL 发生器能力的很小一部分。WorkBook（工作手册）文件包含了很多由那些窗口产生的不同类型的激励。

下面是 SCL 发生器工作手册文件的片段：

```
## SCL Builder Setup File: Do not edit!!

## VERSION: 3.22.00.00
## FORMAT:  v1.40.00
## DEVICE:  PIC24FJ128GA010

## PINREGACTIONS
us
No Repeat
RG12
IC1
--
0
1
1
--
100
1
1
--
150
0
0
--
200
1
1
```

现在一个真正的激励脚本文件可以从刚才定义的定时表里生成。激励脚本文件的扩展名是.scl，这里再次强调，它是简单的 ASCII 文本文件。脚本文件中包含 MPLAB SIM 仿真器用于模拟真实输入信号的命令和信息。激励文件的片段如下：

```
//
// .../IC PS2 simulation.scl
// Generated by SCL Generator ver. 3.22.00.00
// DATE TIME
//


configuration for "pic24fj128ga010" is
end configuration;
```

```
testbench for "pic24fj128ga010" is
begin

 process is
 begin
     wait for 0 us;
     report "Stimulus actions after 0 us";
     RG12 <= '1';
     IC1 <= '1';
     wait;
 end process;

 process is
 begin
     wait for 100 us;
     report "Stimulus actions after 100 us";
     RG12 <= '1';
     IC1 <= '1';
     wait;
 end process;
```

读者可能已经注意到 SCL 文件使用的注释和一些硬件描述语言（VHDL）有一定的相同之处。或者这不仅仅是巧合！

结构化格式的采用，实际上是让激励文件的描述有更大适应性，能更快地执行仿真。

11.2.5 测试 PS/2 接收子程序

在使用产生的激励文件前，需要完成项目的最后一些工作。现在需要把 PS/2 接收子程序打包成"PS2IC.c"模块。记得要把文件放入到项目中（在编辑窗口中单击右键，选择"AddToProject"）。

此外，还要准备一个 include 文件，以发布访问函数 initKBD()、标志位 KBDReady 和用于接收按键码的缓冲器 KDBCode：

```
/*
**
**  PS2IC.h
**
** PS/2 keyboard input library using input capture
*/

extern volatile int KBDReady;
extern volatile unsigned char KBDCode;

void initKBD( void);
```

注意，在这里不需要加入 PS2 接收器实现的其他细节。这使得读者可以随意使用不同的方法而不需要更换接口。将它保存为"PS2IC.h"，并加入到项目中。

下面要生成一个新的文件"PS2ICTest.c"，它包含主程序，并使用 PS2IC 模块测试其性能：

```
/*
** PS2 KBD Test
**
```

```
*/

#include <p24fj128ga010.h>

#include "PS2IC.h"

main()
{
    TRISA = 0xff00;
    initKBD();                   // call the initialization routine

    while ( 1)
    {
        if ( KBDReady)           // wait for the flag
        {
            PORTA = KBDCode;     // fetch the key code and publish on PORTA
            KBDReady = 0;        // clear the flag
        }
    } // main loop
} //main
```

该程序将 POART LSB 初始化为输出（在 Explorer16 上连接到 LED），并且调用 PS/2 键盘初始化子程序，初始化所有选定的输入引脚、状态机以及输入捕捉中断。

主循环会等待中断子程序把标志位变为高（按键码有效），获取按键码，并在 LED 显示器上显示，最后清除标志位，准备接收新的字符。

现在记得把文件加入到项目，然后按“Build All”。

11.2.6　仿真

现在并不急于马上进行仿真实验，而是再次进入调试菜单，选择“Stimulus Controller”子菜单，如图 11-7 所示。

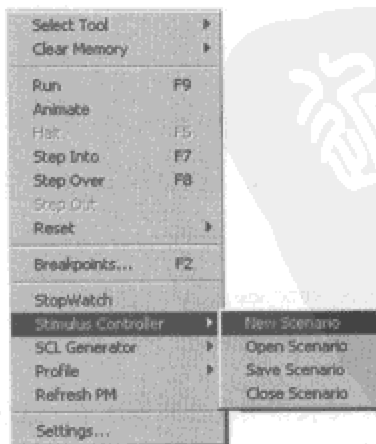


图 11-7　激励控制器子菜单

选择“New Scenario”，然后屏幕上会出现一个新的对话框，如图 11-8 所示。这个就是激励控制器，尽管它有点像 SCL 发生器对话框，不过可不要被它蒙骗了！

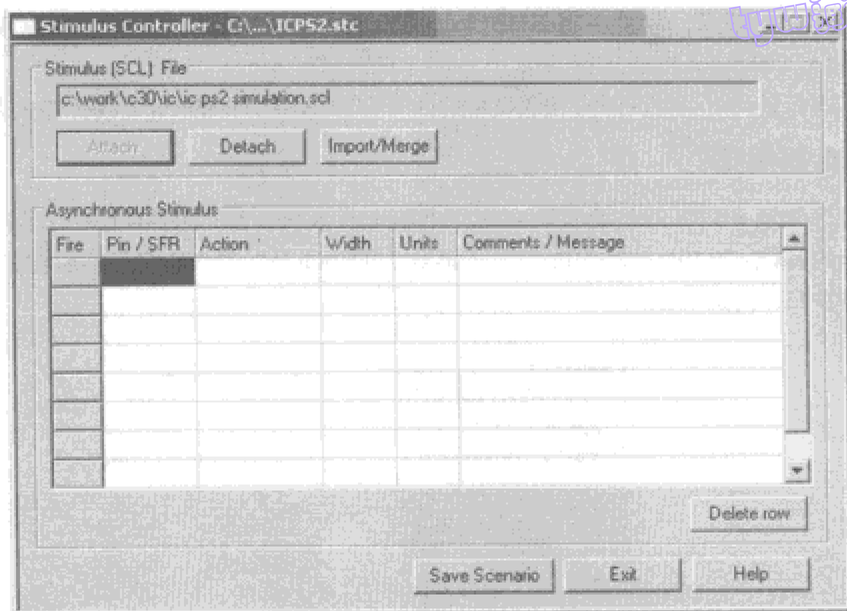


图 11-8 激励控制器窗口

激励控制器可以让用户将 SCL 发生器生成的同步激励脚本连接到项目上，然后通过“Fire”按钮触发“asynchronous stimuli（异步激励）”，就会看到激励控制器表。

选择“Attach”按钮，然后选择之前生成的.SCL 文件。

读者应该将这个“scenario”保存下来，不过在这里，因为将会只处理该.SCL 文件，不需要更进一步的异步激励，所以它其实并没有什么作用。

> 注解 读者应该保持激励控制器窗口为打开状态（在后台）。不要按下“Exit（退出）”键，因为那样就会关闭 scenario 并且不再有激励。

最后一步啦！按下“Reset”（或者选择“Debugger→Reset”），观察当微秒 0 触发启动时，第一个激励发生（如图 11-9 所示）。记住，根据设定的仿真时间表，RG12 和 IC1 线都应该是高的。输出窗口会显示一条确认信息。

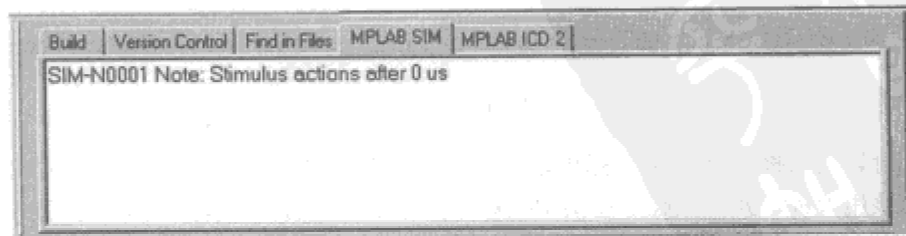


图 11-9 输出窗口（MPLAB SIM 栏）显示激励动作已经触发

现在读者可以选择是单步运行还是全程序运行来验证正确的执行。本书的建议是，读者可

以先在主循环里设置断点，就是在将 KDBCode 复制给 PORTA 的那个位置。打开 Watch 窗口，从 SFR 列表加入 PORTA，然后运行。

几秒后，运行就停止在断点处，PORTA 的内容应该反映仿真的 PS/2 线传输的数据：0x79！

11.2.7 仿真器规范

如果读者想知道 PIC24 的仿真实验在自己电脑上有多快，那么在 MPLAB SIM 调试器菜单上有一个有趣的功能：Profile。选择 Profile 子菜单（"Debugger→Profile"），然后首先单击"Reset Profile"。（如图 11-10 所示。）

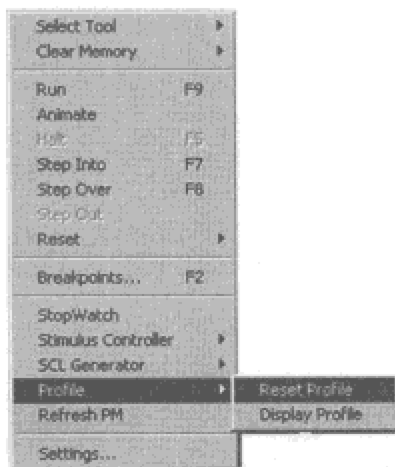


图 11-10 仿真器规范子菜单

这样，仿真器规范计数器和定时器都会清零。然后移除所有的断点，让仿真器运行几秒（"Debugger→Run"）。暂停仿真器，回到"Debugger→Profile"子菜单。这次，选择"Display Profile（显示规范）"。（如图 11-11 所示。）

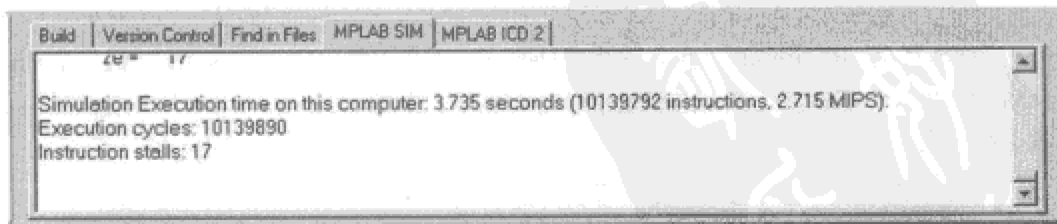


图 11-11 仿真器规范输出

一个相对较长的报告会出现在输出窗口（MPLAB SIM 栏），它列出了在仿真期间处理器对每条指令的使用次数，并且在底部提供了绝对仿真速度的评估。在我的电脑上，显示的是相当大的 2.7 MIPS，即软件仿真（在笔记本上）运行速度大概是真实处理器的 1/6。一点也不差啊！

11.2.8 另一种方法——变化通知

在验证了输入捕捉技术的正常运行后，还有另一种有效连接 PS/2 键盘的方法值得探究。特

别地，PIC24 还有一个有趣的外设可以代替 PS/2 接口，即变化通知（CN）模块。该模块有 22 个 I/O 引脚，可以让用户自己选择适合 PS/2 接口的输入引脚，只要不跟项目或者 Explorer16 已经使用的其他的函数混淆就可以。CN 模块只有四个相关的控制寄存器。CNEN1 和 CNEN2 寄存器包含了每个 CN 输入引脚的中断使能控制位。设置任何其中一位都会对相应引脚的 CN 中断使能。注意只有一个中断向量是对整个 CN 模块有效的，因此它负责中断服务子程序，决定哪个使能的输入已经变化。

表 11-3 CN 控制寄存器表

文件名	Addr	Bit15	Bit14	Bit13	Bit12	Bit11	Bit10	Bit9	Bit8	Bit7	Bit6	Bit5	Bit4	Bit3	Bit12	Bit1	Bit0	全部复位
CNEN1	0060	CN15IE	CN14IE	CN13IE	CN12IE	CN11IE	CN10IE	CN9IE	CN8IE	CN7IE	CN6IE	CN5IE	CN4IE	CN3IE	CN2IE	CN1IE	CN0IE	0000
CNEN2	0062	—	—	—	—	—	—	—	—	—	—	CN21IE	CN20IE	CN19IE	CN18IE	CN17IE	CN16IE	0000
CNPU1	0068	CN15PUE	CN14PUE	CN13PUE	CN12PUE	CN11PUE	CN10PUE	CN9PUE	CN8PUE	CN7PUE	CN6PUE	CN5PUE	CN4PUE	CN3PUE	CN2PUE	CN1PUE	CN0PUE	0000
CNPU2	006A	—	—	—	—	—	—	—	—	—	—	CN21PUE	CN20PUE	CN19PUE	CN18PUE	CN17PUE	CN16PUE	0000

注：— =未使用，读为“0”。复位值以十六进制表示。

每个 CN 引脚还连接有一个弱上拉电阻。该上拉电阻作为电源连接到引脚，并且当有键或者键盘连接时减少对外部电阻的要求。上拉电阻的使能分别由 CNPU1 和 CNPU2 寄存器控制，此外，它们还包含每个 CN 引脚的控制位。设置任何的控制位都会对相应引脚的弱上拉电阻使能。

实际上，PS/2 接口需要的只是将其中一个 CN 输入连接到 PS2 时钟线。在这个例子中，PIC24 的弱上拉电阻并不需要，因为键盘中已经有了。

一共有 22 个候选引脚，而要选的一个 CN 输入不能是模数转换器的引脚（记住需要 5V 引脚），也不能重叠 Explorer16 其他外设的引脚。这就需要查找一下设备数据表和 Explorer16 用户指南。不过一旦选定了输入引脚，例如 CN11（与 PORTG 引脚 9、SPI2 模块的 SS 线还有 PMP 模块的地址线 2 复用），那么新的初始化子程序就只需要几行代码：

```
#DEFINE PS2CLOCK  _RG9    // CN11 input pin
#define PS2DAT    _RG12   // any available 5V tolerant input

void initKBD( void)
{  // PS/2 keyboard
    CNEN1 = 0x0800;        // enable CN11 input change notification
    _CNIF = 0;             // clear the interrupt flag
    _CNIE = 1;             // enable the interrupt on change notification
} // initKBD
```

对于每个中断服务子程序，都可以采用与前面例子相同的状态机，只需加入两行代码以确认正在检查时钟线的下降沿，如图 11-12 所示。

实际上，在使用输入捕捉模块的时候，可以选择只在指定的时钟沿接收中断，而变化通知模块在下降沿和上升沿都会生成中断。在进入中断服务子程序的时候简单地检查一下时钟线的状态，就能区分是哪个边沿。

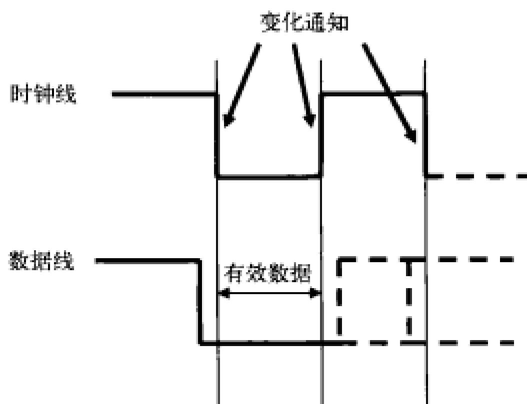


图 11-12　PS/2 接口位定时，变化通知时间信息

```
void _ISR _CNInterrupt( void)
{ // change notification interrupt service routine

// make sure it was a falling edge
    if ( PS2CLK == 0)
{
// PS/2 receiving state machine
  switch( PS2State){
        default:
        case PS2START:
            if ( ! PS2DAT)
            {
                KCount = 8;              // init bit counter
                KParity = 0;             // init parity check
                PS2State = PS2BIT;
            }
            break;

        case PS2BIT:
            KBDBuf >>=1;                 // shift in data bit
            if ( PS2DAT)
                KBDBuf += 0x80;
            KParity ^= KBDBuf;           // update parity
            if ( --KCount == 0)          // if all bit read, move on
                PS2State = PS2PARITY;
            break;

        case PS2PARITY:
            if ( PS2DAT)
                KParity ^= 0x80;
            if ( KParity & 0x80)         // if parity is odd, continue
                PS2State = PS2STOP;
            else
                PS2State = PS2START;
            break;

        case PS2STOP:
            KBDBuf >>=1;                 // shift in data bit
```

```
                if ( PS2DAT)
                    KBDBuf += 0x80;
                KParity ^= KBDBuf;          // update parity
                if ( --KCount == 0)         // if all bit read, move on
                    PS2State = PS2PARITY;
                break;
} // switch state machine

    } // if falling edge

// clear interrupt flag
    _CNIF = 0;

} // CN Interrupt
```

添加前面例子中出现过的常量和变量定义：

```
#include <p24fj128ga010.h>
#include "PS2CN.h"

#define PS2DAT   _RG12        // PS2 Data input pin
#define PS2CLK   _RG9         // PS2 Clock input pin

// definition of the keyboard PS/2 state machine
#define PS2START    0
#define PS2BIT      1
#define PS2PARITY   2
#define PS2STOP     3

// PS2 KBD state machine and buffer
int PS2State;
unsigned char KBDBuf;
int KCount, KParity;

// mailbox
volatile int KBDReady;
volatile unsigned char KBDCode;
```

把所有文件打包成一个文件“PS2CN.c”。

include 文件“PS2CN.h”和前面例子中的几乎是一样的，因为使用的是相同的接口：

```
/*
**
**  PS2CN.h
**
**  PS/2 keyboard input module using Change Notification
*/

extern volatile int KBDReady;
extern volatile unsigned char KBDCode;

void initKBD( void);
```

生成新项目“PS2CN”，并加入.c 和.h 文件。

最后，生成测试该新技术的主模块。再次说明，它和前一个项目几乎是完全一样的：

```
/*
** PS2 KBD Test
**
*/

#include <p24fj128ga010.h>

#include "PS2CN.h"

main()
{
    TRISA = 0xff00;
    initKBD();                  // call the initialization routine

    while ( 1)
    {
        if ( KBDReady)          // wait for the flag
        {
            PORTA = KBDCode;    // fetch the key code and publish on PORTA
            KBDReady = 0;       // clear the flag
        }
    } // main loop
} //main
```

保存项目，然后构建项目（“Project→BuildAll”）以编译和连接所有模块。

要测试变化通知技术，需要再次用到 MPLAB SIM 激励发生性能和重复前一个项目的大部分操作。首先使用 SCL 发生器（“Debugger→SCLGenerator”），生成新的 Workbook（工作手册）。在发生器窗口中，生成两栏，其中一栏用于连接 RG12 的同样的 PS2 数据线，不过另一栏这次是用于连接 CN11 变化通知模块输入的 PS2 时钟线。

在表中添加和上一个例子相同的时间序列和事件，用 CN11 栏代替 IC1 输入栏。将工作手册保存为“`PS2CN.sbs`”，然后单击“Generate SCL”文件生成输出激励脚本文件：“`PS2CN.scl`”。最后，激活激励控制器（“Debugger→StimulusController”），生成新的 Scenario。在激励控制窗口，单击“Attach”，选择“`PS2CN.scl`”文件触发输入仿真。如果需要，可以保存 Scenario，不过不要关掉控制器窗口（尽管可以将它最小化）。

现在可以运行程序代码，测试（在仿真中）新 PS/2 接口的功能了。打开 Watch 窗口，加入 PORTA。在主循环中，将按键码复制到 PORTA 寄存器的下一语句处设置断点。最后，执行复位（“Debugger→Reset”），确定第一个事件已经触发（在 0 μs 设置 PS/2 的两条输入线为高）。运行代码（“Debugger→RUN”），如果一切顺利，不到一秒钟就会看到处理器在断点处停止，然后 PORTA 的内容变成了按键码 0x79。成功！

11.2.9 开销计算

将输入捕捉换成是变化通知的方法实在是太简单了。两个外设都相当强大，尽管是为不同用途设计的，但是任务的处理几乎相同。不过，在嵌入式世界，应该不停地考虑自己是否用最少的资源解决了问题，即使这一目了然，例如在这个例子里，看起来似乎很多余。现在，就来算算每种方法所用资源的真实开销吧。当使用输入捕捉方法时，实际上是使用了 PIC24FJ128GA010 模型 5 个 IC 模块中的一个。该外设是设计成与定时器（Timer2 或者 Timer3）

一起使用的，尽管在示例应用中没有用到时序信息，只用到了与输入沿触发器有关的中断机制。当使用变化通知方法时，只用到了 22 个引脚中的一个引脚，却控制了该外设的唯一中断向量。换而言之，如果还需要变化通知模块去控制其他的输入引脚，那么就要共用中断向量，增加方法的延时和复杂度。这两个方法不分胜负。

11.2.10 第三种方法——I/O 查询

PS/2 键盘接口还有一种需要介绍的方法。它是最基本的方法，并且需要使用一个定时器，设置周期性的中断，而输入可以是微控制器上任何（5V 的）I/O 引脚，如图 11-13 所示。因此，这种方法从结构和设计上看来都是最灵活的。它也是最常见的，因为任何微控制器模型，即使是最小和最廉价的，至少都有一个定时器模块可以满足它的需求。理论上它的操作也十分简单。只要设置好定时器对应的周期寄存器的值，那么按照一定的时间，它就会产生一个中断。

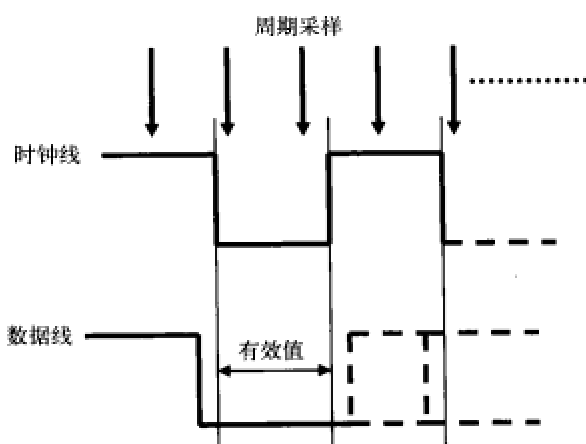


图 11-13 PS/2 接口位定时，I/O 查询采样点

这次要使用以前从未用过的 Timer4。PR4 是周期寄存器。中断服务子程序（T4 中断）会对 PS/2 时钟线采样，然后判断在前一个周期中 PS/2 时钟线是否出现下降沿。当检测到下降沿时，数据线的状态就会被认为是接收到按键码。为了确定采样的频率，得到 PR4 寄存器的最优值，可以检查 PS/2 时钟线相邻边缘的最短间隔时间。这是由 PS/2 接口的最大波特率决定的，根据经验该值大约为 16 Kb/s。在该速率下，时钟信号可以用占空比大约为 50%的方波表示，周期大概是 62.5 μs。换而言之，当数据线上有一位数据时，时钟线的低电平持续时间要略大于 30 μs，然后高电平保持相同的时间，直到数据位被移出。要使得中断时间小于 30 μs（比如是 25 μs），需要保证时钟线在两个连续的边沿间至少有一次的采样。键盘的传输波特率只有 10 Kb/s，因此两个边沿的最大间隔为 50 μs。所以，在每个时钟沿之间应该对时间和数据线采样两次，甚至 3 次。换而言之，需要建立一个新的状态机，以检测下降沿准确的出现时间，并且保持对 PS/2 时钟信号的正确追踪，如图 11-14 所示。

状态机只需要两种状态，所有的转换如表 11-4 所示。

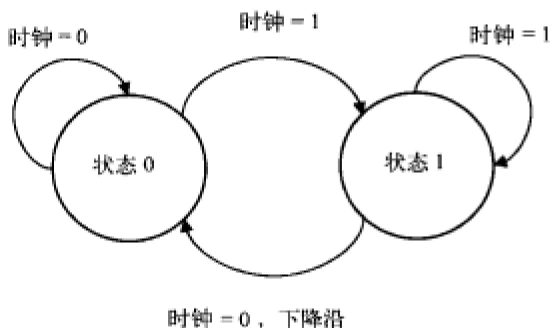


图 11-14 时钟查询状态机图示

表 11-4 时钟检测状态机转换表

状 态	条 件	结 果
状态 0	时钟=0	保持状态 0
	时钟=1	上升沿，转到状态 1
状态 1	时钟=1	保持状态 1
	时钟=0	检测到下降沿 执行数据状态机 转到状态 0

一旦检测到下降沿，就可以使用前面项目中的状态机来读取数据线。需要特别注意的是，在这里，数据线的值并不是在时钟线的下降沿发生后立即采样，而是有一定的延时。为了避免读取到数据线上有效值以外的值，必须对时钟和数据线同时采样。根据定义（PS/2 特性），如果时钟线为低，数据就必须视为有效。实际上，需要将数据和时钟输入分配给相同端口的引脚。在这个例子中，会选择 RG12 作时钟线，RG15 作数据线。这样，一旦进入中断服务子程序，就可以将 PORTG 的内容赋值给临时变量，一个小动作就可以完美地实现对两条线的同步采样。以下的代码就能完成图 11-14 所示的最简单时钟状态机：

```
#define PS2DAT   _RG12        // PS2 Data input pin
#define PS2CLK   _RG15        // PS2 Clock input pin
#define CLKMASK  0x8000       // mask to detect the clock line
#define DATMASK  0x1000       // mask to detect the data line


unsigned char KBDBuf;
int KState;


// mailbox
volatile int KBDReady;
volatile unsigned char KBDCode;

void _ISR _T4Interrupt( void)
{
    int PS2IN;
```

```
// sample the inputs, clock and data, at the same time
PS2IN = PORTG;

// Keyboard clock state machine
if ( KState)
{   // previous time clock was high, State1
    if ( !(PS2IN & CLKMASK))        // PS2CLK = 0

{// falling edge detected
        KState = 0;                 // transition to State0

        <<<... Insert Data state machine here!>>>


    } // falling edge

    else
    { // clock still high, remain in State1

    } // clock still high
} // State 1

else
{ // State 0
    if ( PS2IN & CLKMASK)           // PS2CLK = 1
    { // rising edge detected
        KState = 1;             // transition to State1

    } // rising edge

    else
    { // clocl still low, remain in State0

    } // clock still low
} // State 0

// clear the interrupt flag
_T4IF = 0;

} // T4 Interrupt
```

有了这个刚刚建立的周期性查询机制，就可以为 PS2 接口加入新的功能，让它以最小的开销赢得更大的鲁棒性。首先，可以向时钟状态机的两个状态的空闲循环增加一个计数器。这样就有了一个暂停时间，可以检测并修正 PS/2 键盘在传送阶段的断开状态或者接收子程序丢失同步信号的错误状态。

带超时计数器 Ktimer 的新的状态转换表可更新为表 11-5。

表 11-5 时钟查询（带超时）状态机转换表

状　态	条　件	结　果
状态 0	时钟=0	保持状态 0 KTimer 减 1

（续）

状　态	条　件	结　果
状态 0	时钟=0	如果 Ktimer=0，错误 重启数据状态机
	时钟=1	上升沿，转到状态 1
状态 1	时钟=1	保持状态 1 KTimer 减 1 如果 Ktimer=0，错误 重启数据状态机
	时钟=0	检测到下降沿 执行数据状态机 转到状态 0 重启 Ktimer

新的转换表只需向原来的中断服务子程序加入几行指令：

```
void _ISR _T4Interrupt( void)
{
    int PS2IN;

    // sample the inputs, clock and data, at the same time
    PS2IN = PORTG;

    // Keyboard clock state machine
    if ( KState)
    {   // previous time clock was high, State1
        if ( !(PS2IN & CLKMASK))        // PS2CLK = 0
        {// falling edge detected
            KState = 0;                 // transition to State0
            KTimer = KMAX;              // restart the counter

          <<<... Insert Data state machine here!>>>


        } // falling edge

        else
        { // clock still high, remain in State1
            KTimer--;
            if ( KTimer ==0)            // timeout!
                PS2State = PS2START;    // reset the data state machine

        } // clock still high
    } // State 1

    else
    { // State 0
        if ( PS2IN & CLKMASK)           // PS2CLK = 1
        { // rising edge detected
            KState = 1;                 // transition to State1

        } // rising edge
```

```
            else
            { // clocl still low, remain in State0
                KTimer--;
                if ( KTimer == 0)           // timeout!
                    PS2State = PS2START;    // reset the data state machine
            } // clock still low
        } // State 0

        // clear the interrupt flag
        _T4IF = 0;

    } // T4 Interrupt
```

11.2.11 测试 I/O 查询方法

首先，要从前一个项目中插入数据状态机，修改为在中断服务子程序入口的 PS2IN 中采样：

```
switch( PS2State){
default:
case PS2START:
    if ( !(PS2IN & DATMASK))
    {
        KCount = 8;             // init bit counter
        KParity = 0;            // init parity check
        PS2State = PS2BIT;
    }
    break;

case PS2BIT:
    KBDBuf >>=1;                        // shift in data bit
    if ( PS2IN & DATMASK)               //PS2DAT
        KBDBuf += 0x80;
    KParity ^= KBDBuf;                  // calculate parity
    if ( --KCount == 0)                 // if all bit read, move on
        PS2State = PS2PARITY;
    break;

case PS2PARITY:
    if ( PS2IN & DATMASK)
        KParity ^= 0x80;
    if ( KParity & 0x80)                // if parity is odd, continue
        PS2State = PS2STOP;
    else
        PS2State = PS2START;
    break;

case PS2STOP:
    if ( PS2IN & DATMASK)               // verify stop bit
    {
        KBDCode = KBDBuf;       // write in the buffer
        KBDReady = 1;                   // set flag
    }
    PS2State = PS2START;
    break;

} // switch
```

下面使用正确的初始化子程序来完成第三个模块：

```
void initKBD( void)
{
    // init I/Os
    _TRISG15 = 1;   // make RG15 an input pin, PS/2 Clock
    _TRISG12 = 1;   // make RG12 an input pin, PS/2 Data

    // clear the flag
    KBDReady = 0;

    PR4 = 25 * 16;  // 25 us, set the period register
    T4CON = 0x8000; // T4 on, prescaler 1:1
    _T4IF = 0;      // clear interrupt flag
    _T4IE = 1;      // enable interrupt

} // init KBD
```

这样编写的程序简洁明了。

将所有程序保存在一个模块“PS2T4.c”中。然后生成一个 include 文件：

```
/*
**
**   PS2T4.h
**
**   PS/2 keyboard input library using T4 polling
*/

extern volatile int KBDReady;
extern volatile unsigned char KBDCode;


void initKBD( void);
```

这跟前面模块的 include 文件完全相同，并且主模块部分也没有太大的区别：

```
/*
** PS2 KBD Test
**
**
*/

#include <p24fj128ga010.h>

#include "PS2T4.h"

main()
{
    TRISA = 0xff00;
    initKBD();                 // call the initialization routine

    while ( 1)
    {
        if ( KBDReady)         // wait for the flag
        {
```

```
            PORTA = KBDCode;     // fetch the key code and publish on PORTA
            KBDReady = 0;              // clear the flag
        }
    } // main loop
} //main
```

生成新项目“PS2T4”并把 3 个文件添加进去。编译并遵循与前面两个例子相同的步骤生成激励脚本文件“PS2T4.scl”。要记住，这次时钟线的激励由 RG15 引脚提供。打开新的 Scenario 以及新的激励控制器和激励脚本文件开始仿真实验（记住要让激励控制器窗口保持在后台打开）。打开 Watch 窗口，加入 PORTA。最后在 PORTA 的赋值语句后面设置断点并运行。如果一切正常，这次，还是会在 Watch 窗口中看到 PORTA 的值变成 0x79。又成功了！

11.2.12 方案性价比

对比前两个例子，可以看到 I/O 查询方法为用户提供了自由度最大的输入引脚选择，并且只需要一个电源、一个定时器和一个中断向量。周期性的中断可以无缝地共享其他任务，以形成共同的时间基数，前提是它们都简化为几个查询周期。超时特性是附加的功能，如果要将它应用于之前的方法，那么除了使用输入捕捉或变化通知模块以及中断，还必须使用一个分立的定时器和另一个中断服务子程序。至于代码的有效性，输入捕捉和变化通知模块看起来更有优势，因为中断只在检测到边沿的时候才发生。实际上，正如读者所见，输入捕捉是符合这种评价观点的，因为可以由用户明确地指定边沿——也就是时钟线的下降沿。表面上看 I/O 查询方法需要最长的中断服务子程序，不过代码的行数并不能真正地反映出中断服务子程序的大小。实际上，I/O 查询中断服务子程序中的两个嵌套状态机里，只有其中的几行指令会在每次调用中用到，因此执行时间很短，开销也最小。

为了验证中断服务子程序的实际软件开销，可以对 PS/2 接口的三种方法都进行一个简单的测试。以最后一种方法为例。可以使用一个 I/O 引脚（逻辑上是选用 LED 输出引脚）来帮助实现中断服务子程序中微控制器的可视化。可以将这个引脚放在开始的位置，并在退出前复位：

```
void _ISR _T4Interrupt( void)
{
    _RA0 = 1;                    // flag up, inside the ISR
    ...

    <<< Interrupt service routine here >>

    _RA0 = 0;                    // flag down, back to the main
}
```

使用 MPLAB SIM 仿真器的逻辑分析器视图，就可以在计算机显示屏上看到它。按照逻辑分析器的列表，使能追踪缓冲器，并设置正确的仿真速度。选择 RA0 通道，重构项目。如果要测试前两种方法，就需要再次激活激励控制器以生成输入信号，否则，将没有中断产生。

要测试查询子程序，则不需要激励。定时器中断总是会发生的，这里要测试的，是在没有键盘输入的情况下，连续的查询将耗费多少时间。

让 MPLAB SIM 执行几秒钟，然后停止仿真器，转回逻辑分析器窗口。读者可以将窗口放大以得到适当的视图，如图 11-15 所示。

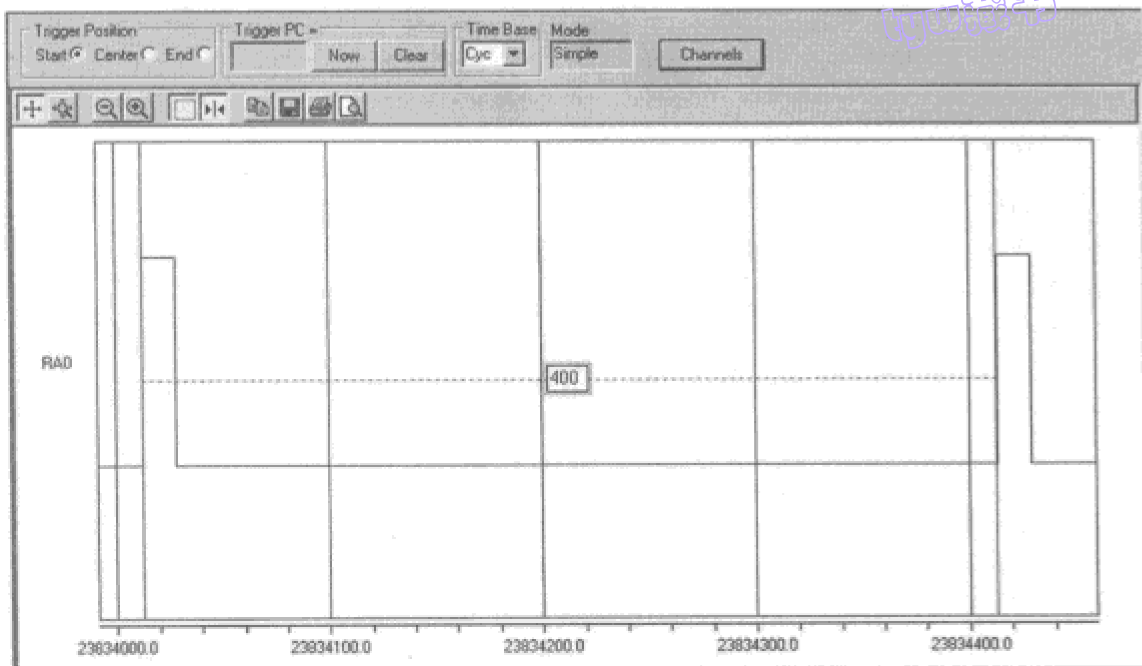


图 11-15 逻辑分析器视图，测量 I/O 查询周期

激活光标▸|◂，然后测量两个相邻 RA0 上升沿间隔的周期数。由于使用的是 25 μs 周期，可以看到两个调用之间有 400 个周期（25 μs*16 个周期/μs @32 MHz）。通过测量 RA0 上升沿与下降沿之间的周期数，就可以大概知道在中断服务子程序中消耗的时间，在这里我找到的是 16 个周期。两个数值的比值反映出 PS/2 接口消耗了多少的计算机功率。在这个例子中，看到它只有 2.5%。

11.2.13 完成接口：添加 FIFO 缓冲器

除了目前介绍的三种方法外，在完成 PS/2 键盘模块接口之前，还需要来探讨一些细节问题。首先，需要在 PS/2 接口程序和“消费者”或者其他主要应用之间加入一个 FIFO 缓冲器。到目前为止，实际上，只用到了一个简单的邮箱机制来保存接收到的最后一个按键码。如果更深入地探究 PS/2 键盘协议，就会发现，当一个按键被按下又释放时，至少有 3 个（最多是 5 个）按键码会被送入主机。如果将 shift、control 和 Alt 三键同时按下，那么事情会变得更为复杂一些，读者马上就会发现单字节的邮箱是不够用的。在实践中，建议使用至少 16 字节的 FIFO 缓冲器。缓冲器的输入可以是同接收中断服务子程序的简单组合，当接收一个新的按键码时，就立即插入到 FIFO 缓冲器中。缓冲器可以定义成字符数组，并且使用两个指针就可以从头到尾循环地搜索缓冲区，如图 11-16 所示。

```
// circular buffer
unsigned char KCB[ KB_SIZE];

// head and tail or write and read pointers
volatile int KBR, KBW;
```

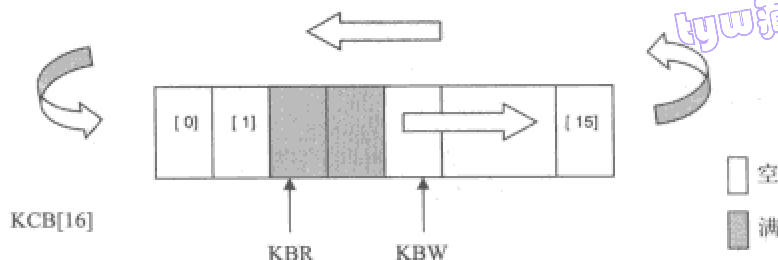


图 11-16 环状 FIFO 缓冲器

根据以下几条简单的规则，就可以对缓冲器的内容进行搜索。

- 写指针 KBW（或头），表明第一个空位置，可以接收下一个按键码。
- 读指针 KBR（或尾），表明第一个满位置。
- 当缓冲器为空，KBR 和 KBW 都指向同一位置。
- 当缓冲器为满，KBW 指向 KBR 前的位置。
- 当从缓冲器读取或写入一个字节，对应的指针加 1。
- 当到达数组的最后，每个指针都会回到数组的第一个元素。

将下面的代码插入初始化子程序中：

```
// init the circular buffer pointers
KBR = 0;
KBW = 0;
```

然后更新中断服务子程序状态机的 STOP 状态：

```
case PS2STOP:
        if ( PS2IN & DATMASK)        // verify stop bit
        {
            KCB[ KBW] = KBDBuf;      // write in the buffer
            if ( (KBW+1)%KB_SIZE != KBR) // check if buffer full
                KBW++;               // else increment buffer
            KBW %= KB_SIZE;          // wrap around
        }
        PS2State = PS2START;
        break;
```

其中，“%”运算符是指除以缓冲器大小的余数，可以保证指针在环状缓冲器中回到开始位置。

从 FIFO 缓冲器中获取按键码需要考虑几个问题。具体来说，当选择的是输入捕捉或者变化通知方法时，需要加入一个新的函数（getKeyCode()）来代替邮箱/标志位机制。如果缓冲器中没有可用的按键码，函数将会返回 FALSE；如果缓冲器中至少有一个按键码，那么代码就会通过指针返回：

```
int getKeyCode( char *c)
{
    if ( KBR == KBW)             // buffer empty
    return FALSE;

    // buffer contains at least one key code
    *c = KCB[ KBR++];            // extract the first key code
```

```
    KBR %= KB_SIZE;                       // wrap around the pointer

    return TRUE;
} // getKeyCode
```

注意，提取子程序只改变读指针，因此，应该在中断有效的时候才执行这个操作。假设在提取阶段会出现中断，那么有以下两个可能性。

- 缓冲器为空：一个新的按键码会加入，但是 getKeyCode 子程序只会在下次调用时“通知”有效字符。
- 缓冲器非空：如果还有空间，那么中断子程序会将新字符添加到缓冲器的尾部。

在两种情况下，都不需要特别考虑冲突或者危险结果。

如果选择的是查询方法，那么还需要考虑一个问题。实际上，由于定时器中断是一直有效的，因此可以利用它来执行更多的任务。方法就是保持简单的“邮箱-标志位”机制来作为传送按键码的接收子程序代码的接口，同时使用中断不断地检查邮箱，随时用 FIFO 缓冲器的内容来补充。这样就可以把整个 FIFO 的管理交给中断服务子程序负责，让缓冲器完全透明，并保持邮箱传递接口的简单。I/O 查询机制的新的完整中断服务子程序如下：

```
void _ISR _T4Interrupt( void)
{
    int PS2IN;
    // check if buffer available
    if ( !KBDReady && ( KBR!=KBW))
    {
        KBDCode = KCB[ KBR++];
        KBR %= KB_SIZE;
        KBDReady = 1;                // signal character available
    }

    // sample the inputs clock and data at the same time
    PS2IN = PORTG;

// Keyboard state machine
if ( KState)
{   // previous time clock was high KState 1
    if ( !(PS2IN & CLKMASK))      // PS2CLK = 0
    {   // falling edge detected,
        KState = 0;              // transition to State0
        KTimer = KMAX;           // restart the counter

        switch( PS2State){
        default:
        case PS2START:
            if ( !(PS2IN & DATMASK))
            {
                KCount = 8;          // init bit counter
                KParity = 0;         // init parity check
                PS2State = PS2BIT;
            }
            break;

        case PS2BIT:
            KBDBuf >>=1;             // shift in data bit
```

```
            if ( PS2IN & DATMASK)          //PS2DAT
                KBDBuf += 0x80;
            KParity ^= KBDBuf;             // calculate parity
            if ( --KCount == 0)            // if all bit read, move on
                PS2State = PS2PARITY;
            break;

        case PS2PARITY:
            if ( PS2IN & DATMASK)
                KParity ^= 0x80;
            if ( KParity & 0x80)           // if parity is odd, continue
                PS2State = PS2STOP;
            else
                PS2State = PS2START;
            break;

        case PS2STOP:
            if ( PS2IN & DATMASK)               // verify stop bit
            {
                KCB[ KBW] = KBDBuf;             // write in the buffer
                if ( (KBW+1)%KB_SIZE != KBR) // check if buffer full
                    KBW++;                      // else increment buffer
                KBW %= KB_SIZE;                 // wrap around
            }
            PS2State = PS2START;
            break;

            } // switch
        } // falling edge
        else
        { // clock still high, remain in State1
            KTimer--;
            if ( KTimer ==0)
                PS2State = PS2START;
        } // clock still high
    } // Kstate 1
    else
    { // Kstate 0
        if ( PS2IN & CLKMASK)      // PS2CLK = 1
        { // rising edge, transition to State1
            KState = 1;
        } // rising edge
        else
        { // clocl still low, remain in State0
            KTimer--;
            if ( KTimer == 0)
                PS2State = PS2START;
        } // clock still low
    } // Kstate 0

    // clear the interrupt flag
    _T4IF = 0;

} // T4 Interrupt
```

11.2.14　完成接口：解码按键码

到目前为止，还没有介绍过按键码，可能读者会认为它们和每个按键的 ASCII 码是匹配的，例如，当在键盘上按下"A"，那么发送的就是 ASCII 码（0x41）。然而，事实并非如此。由于历史原因，即使是最新款的 USB 键盘使用的仍然是"扫描码"，也就是每一个按键分配的数值可以追溯到 1980 年左右，第一个 IBM PC 键盘所使用的最初的（8048 微控制器）键盘扫描固件。实际上，将按键码转换为特定字符集通常出现在更高级场合（由 Windows 键盘驱动执行），这是一件好事，因为这样对于多种国际键盘结构有了通用的机制。还要记住，同样是历史原因，至少有 3 种不同和部分兼容的"扫描码集"。幸好，全部键盘都默认支持扫描码集#2，也就是下面将要关注的。

当每次按键发生（任何键，包括 shift 键或者控制键）时，它对应的扫描码都会被送至主机。这个叫做"通码"。但是同时，当相同的键被释放时，新的扫描码序列也会被送至主机。这个叫做"断码"。断码通常是由相同的扫描码再加上"0xF0"作为前缀构成。有些按键有两字节长的通码（如 Ctrl、Alt 和箭头），因此它们的断码就有 3 个字节长。表 11-6 举例列出了几种扫描码集 #2（默认）的通码和断码。

表 11-6　扫描码集#2（默认）的通码和断码举例

按　键	通　码	断　码
"A"	1C	F0,1C
"5"	2E	F0,2E
"F10"	09	F0,09
右箭头	E0,74	F0,E0,74
右"Ctrl"	E0，14	F0,E0，14

为了处理这个信息，并把扫描码翻译成正确的 ASCII 码，需要有一张表来将基本的扫描码和基本的美式英语键盘对应起来。

```
// PS2 keyboard codes (standard set #2)
const char keyCodes[128]={
                0,   F9,    0,   F5,   F3,   F1,   F2, F12,   //00
                0, F10,   F8,   F6,   F4, TAB,  '`',    0,   //08
                0,    0,L_SHFT,  0,L_CTRL,'q','1',    0,   //10
                0,    0,  'z',  's',  'a',  'w',  '2',    0,   //18
                0,  'c',  'x',  'd',  'e',  '4',  '3',    0,   //20
                0,  ' ',  'v',  'f',  't',  'r',  '5',    0,   //28
                0,  'n',  'b',  'h',  'g',  'y',  '6',    0,   //30
                0,    0,  'm',  'j',  'u',  '7',  '8',    0,   //38
                0,  ',',  'k',  'i',  'o',  '0',  '9',    0,   //40
                0,  '.',  '/',  'l',  ';',  'p',  '-',    0,   //48
                0,    0,'\'',    0,  '[',  '=',    0,    0,   //50
          CAPS, R_SHFT,ENTER,  ']',    0,0x5c,    0,    0,   //58
                0,    0,    0,    0,    0,    0, BKSP,    0,   //60
                0,  '1',    0,  '4',  '7',    0,    0,    0,   //68
                0,  '.',  '2',  '5',  '6',  '8', ESC, NUM,   //70
              F11,  '+',  '3',  '-',  '*',  '9',    0,    0    //78
            };
```

注意，数组被定义为 const，位于程序存储器中，以节省更多的 RAM 空间。

对于每个键的 shift 功能，也应该有一张类似的表可以方便使用。

```
const char keySCodes[128] = {
                  0,  F9,   0,  F5,  F3,  F1,  F2, F12,    //00
                  0, F10,  F8,  F6,  F4, TAB, '~',   0,    //08
                  0,   0,L_SHFT, 0,L_CTRL,'Q','!',   0,    //10
                  0,   0, 'Z', 'S', 'A', 'W', '@',   0,    //18
                  0, 'C', 'X', 'D', 'E', '$', '#',   0,    //20
                  0, ' ', 'V', 'F', 'T', 'R', '%',   0,    //28
                  0, 'N', 'B', 'H', 'G', 'Y', '^',   0,    //30
                  0,   0, 'M', 'J', 'U', '&', '*',   0,    //38
                  0, '<', 'K', 'I', 'O', ')', '(',   0,    //40
                  0, '>', '?', 'L', ':', 'P', '_',   0,    //48
                  0,   0,'\"',   0, '{', '+',   0,   0,    //50

           CAPS, R_SHFT,ENTER, '}',   0, '|',   0,   0,    //58
                  0,   0,   0,   0,   0,   0, BKSP,  0,    //60
                  0, '1',   0, '4', '7',   0,   0,   0,    //68
                  0, '.', '2', '5', '6', '8', ESC, NUM,    //70
                F11, '+', '3', '-', '*', '9',   0,   0     //78
              };
```

对于全部的 ASCII 字符，这种翻译都是很直接的，但是仍然需要给 shift 键和控制键等功能分配一些特殊值。它们中只有一部分能从 ASCII 集中找到相应的代码：

```
// special function characters
#define TAB      0x9
#define BKSP     0x8
#define ENTER    0xd
#define ESC      0x1b
```

对于其他的字符，就需要创建自己的规范，或者在需要使用时，忽略它们并给它们分配一个公共码（0）：

```
#define L_SHFT  0x12
#define R_SHFT  0x12
#define CAPS    0x58
#define L_CTRL  0x0
#define NUM     0x0
#define F1      0x0
#define F2      0x0
#define F3      0x0
#define F4      0x0
#define F5      0x0
#define F6      0x0
#define F7      0x0
#define F8      0x0
#define F9      0x0
#define F10     0x0
#define F11     0x0
#define F12     0x0
```

下面的程序中 getC() 将实现大部分普通代码的基本翻译，并且对 shift 状态和 CAPS 键的翻转特别关注：

```
int CapsFlag=0;

char getC( void)
{
    unsigned char c;

    while( 1)
    {
        while( !KBDReady);        // wait for a key to be pressed
        // check if it is a break code
        while (KBDCode == 0xf0)
        {   // consume the break code
            KBDReady = 0;
            // wait for a new key code
            while ( !KBDReady);
            // check if the shift button is released
            if ( KBDCode == L_SHFT)
                CapsFlag = 0;
            // and discard it
            KBDReady = 0;
            // wait for the next key
            while ( !KBDReady);
        }
        // check for special keys
        if ( KBDCode == L_SHFT)
        {
            CapsFlag = 1;
            KBDReady = 0;
        }
        else if ( KBDCode == CAPS)
        {
            CapsFlag = !CapsFlag;
            KBDReady = 0;
        }

        else // translate into an ASCII code
        {
            if ( CapsFlag)
                c = keySCodes[KBDCode%128];
            else
                c = keyCodes[KBDCode%128];
            break;
        }
    }
    // consume the current character
    KBDReady = 0;

    return ( c);
} // getC
```

11.3　飞后小结

在本章中介绍了 PS/2 计算机键盘的接口及三种操作方法，学习了两个新的外设模块：输入

捕捉模块和变化通知模块。还讨论了实现 FIFO 缓冲器和改进中断管理的方法。在整章的学习中，始终关注的是每一种解决方案的资源消耗和性能之间的权衡问题。

11.4 提示与技巧

禁止键盘的传输——开漏极输出控制

每个 PS/2 键盘内部都有一个深度为 16 按键码的 FIFO 缓冲器。这样即使在主机来不及接收的时候，键盘也可以驻留用户的输入。正如在本章开始的时候提到的，主机可以在任何指定时间将时钟线变低来暂停通信（至少 100 μs），并且保持一个相当长的时间。当时钟线被释放时，键盘恢复传输。如果最后一个按键码被打断，将会从 FIFO 缓冲器中重新读取并发送。

为了能像主机那样阻止键盘传送，需要使用开漏极驱动的输出来控制时钟线。幸好，由于 PIC24 具有特殊的 I/O 端口模块，因此这很容易实现。实际上，每个 I/O 端口（PORTx）都有一个对应的控制寄存器（ODCx），可以独立地控制每个引脚的输出驱动来操作开漏极模式。

注解 这个特性对于 PIC24 对任何 5V 设备的输出接口都是非常有用的。

对于前面的例子，要把 PS/2 时钟线转换成开漏极输出，只需添加下面几行代码：

```
_ODG13 = 1;        // configure the PORTG pin 13 output driver in open-drain
_LATG13 = 1;       // initially let the output in pull up
_TRISG13 = 0;      // enable the output driver
```

注意，和其他 PIC 微控制器一样，即使一个引脚被设置成输出，它的当前状态仍可能读为输入。因此，在暂停和接收键盘字符交替进行的时候，不需要一直转换输出和输入状态。

11.5 练习

(1) 添加一个函数，向键盘发送命令控制 LED 状态并设置按键的重复率。
(2) 将“stdio.h”库输入函数 read () 重定向成来自 stdin 流的键盘输入。
(3) 增加支持 PS/2 鼠标接口。

11.6 推荐书目

- Anderson F.(2003)
  Flying the Mountains
  McGraw-Hill, New York, NY
  飞越高山需要更多的留心和准备。这将会是取得私人飞机驾驶照后的下一个挑战。

11.7 网上链接

- http://www.computer-engineering.org/
  在这个出色的网站里，有很多 PS/2 键盘和鼠标接口方面的有用资料。

第 12 章 暗 屏

本章内容

- ▶产生合成视频信号
- ▶使用输出比较模块
- ▶存储器分配
- ▶图像串行化
- ▶构建视频模块
- ▶测试视频发生器
- ▶性能测量
- ▶暗屏
- ▶测试图样
- ▶描点
- ▶星夜
- ▶画线
- ▶Bresenham 算法
- ▶画数学函数图
- ▶二维函数可视化
- ▶分形几何
- ▶文本
- ▶测试 TextOnGPage 模块
- ▶开发文本页视频
- ▶测试文本页性能

夜间开车是一件很爽的事情。通常路上的车比较少，而且空气比较清爽，除非真的很累，否则对面来车的灯光并不会让人生厌。然而，当第一次受命要进行夜间的跨国飞行时，飞行学员的确有些担心。必须承认的是，看到挡风玻璃外空洞洞的一片漆黑，真是一件令人毛骨悚然的事情。不过，在经过一个星期的真实体验后，飞行学员的态度将完全改变。诚然，夜间飞行比普通的绕圈飞行要严肃很多。这需要更精密的计划，但一切都是值得的。飞过无人居住的区域时，放眼望满天星光，是城市人很难看到的风景——感觉就像是乘坐着宇宙飞船飞往另一个太阳系一样。而在城市上空或者旁边飞过，就会看到停车场和房屋交织成一个壮丽的灯光表演秀——如同圣诞节那样。灯光熄灭了，但屏幕并不是真正的漆黑。这是一个大型的演出，并且每晚都在上演。

12.1 飞行计划

本章将介绍 TV 屏幕或者任何接收标准合成视频信号的显示器的接口技术。趁这个机会，可以使用一些 PIC24 外部模块的新性能以及学习新的编程技巧。第一个项目目标，就是要得到一个完美的暗屏（同步的视频帧），然后再用一些有趣的图形应用来填充它。

12.2 飞行

目前，视频的格式和标准有很多种，或者最传统和最常见的就是所谓的“合成”视频格式。这正是第一台电视机所采用的格式，而现在它代表了所有视频显示的最小交集，无论是最新款的高清平板电视，还是 DVD 播放机，或者 VHS 播放机。所有的视频设备都基于同一个原则：每次只“绘出”一行的图像，从屏幕的左上角开始，平行扫描到右边，然后快速地跳回下一位

置的左边缘，开始画第二行，以此类推，以“Z”字形式，直到整个屏幕都扫描完毕（如图 12-1 所示）。然后再重新开始该处理过程，整个画面以很快的速度更新，以欺骗观众的眼睛，使其相信整个画面是同时呈现的，如果有运动图画的话，那将会是非常流畅和连续的。

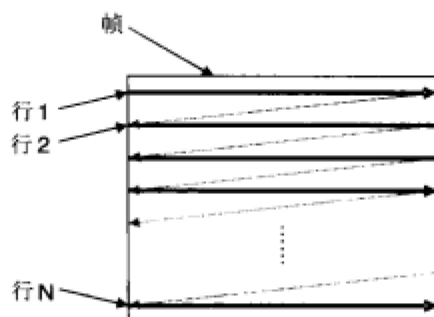


图 12-1 视频图像扫描

在过去几年里，世界上的其他地方出现了稍不兼容的系统，不过基本机制不变。改变的是图像的行数、刷新频率和颜色信息的编码方式。

表 12-1 列出了在美国、欧洲和亚洲使用得最广泛的 3 种视频标准。以上的标准将“亮度”信息（底色的黑白图像）和同步信息编码成简单的合成信号（如图 12-2 所示）。

表 12-1 国际视频标准示例

	美 国	欧洲、亚洲	法国和其他
标准	NTSC	PAL	SECAM
每秒帧数	29.97(1)	25	25
每帧行数	525	625	625

(1) NTSC 在过去是 30 帧/秒，不过在引入新的颜色标准以后改为 29.97，以适应“颜色子载波”晶振的特定频率。

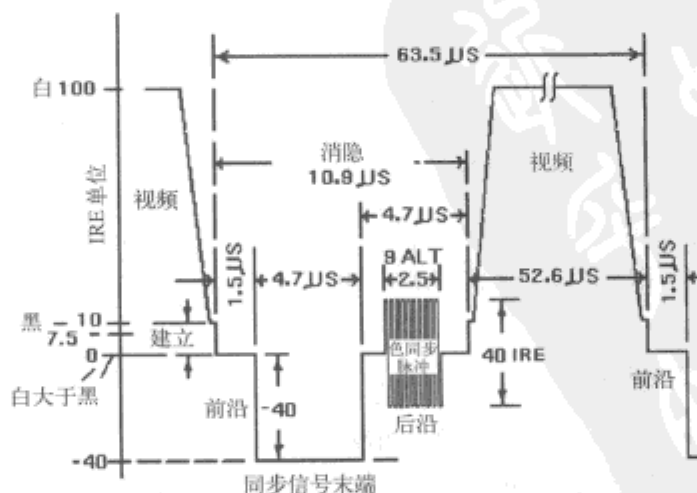


图 12-2 NTSC 合成信号（水平行信号）

名词“合成”说明了一个视频信号是由 3 种信息组合而成的：信号的亮度以及水平方向和垂直方向的同步信息。

实际上，水平行信号由以下的环节组成。

- 水平同步信号，用于确定每一行的开始。
- 后沿，产生图像的暗帧。
- 实际行信号亮度。电压越高，则点越亮。
- 前沿，产生图像的右边沿。

颜色信息单独传送，由高频子载波调制。3 种主要的标准颜色信息的编码方式是有明显区别的，不过，这里将忽略颜色编码带来的问题，只输出简单的黑白图像。

所有的标准都是采用一种叫做“隔行扫描”的技术来以较低带宽获得（相对的）高质量的输出。实际上，每一帧中只有一半的行信息传输到屏幕上显示。相邻的帧只给出图片的奇数行或者偶数行，因此整个图像内容使用刷新速度的一半（PAL 和 NTSC 分别是 25 Hz 和 30 Hz）就可以有效更新。这种方法对于电视广播来说是很有效的，不过对于文本或者水平行的显示，就会产生讨厌的闪烁，正如很多电脑显示器出现的那样。因此，现在的电脑显示都不使用“隔行扫描”，而采用逐行扫描。现在很多电视机，尤其是使用 LCD 和等离子技术的电视，对接收的广播图像都进行隔行扫描解码。在本章的项目中，也避免使用“隔行扫描”，以牺牲一半的图像质量，获得更稳定可靠的显示输出。换而言之，每秒会以双倍速率传输 60 帧、每帧 262 行（NTSC 标准）的信息。接触过 PAL 或者 SECAM 电视机/显示器的读者，就会发现项目很容易就可以改成每秒 50 帧、每帧 312 行。

一个完整的视频帧信号如图 12-3 所示。

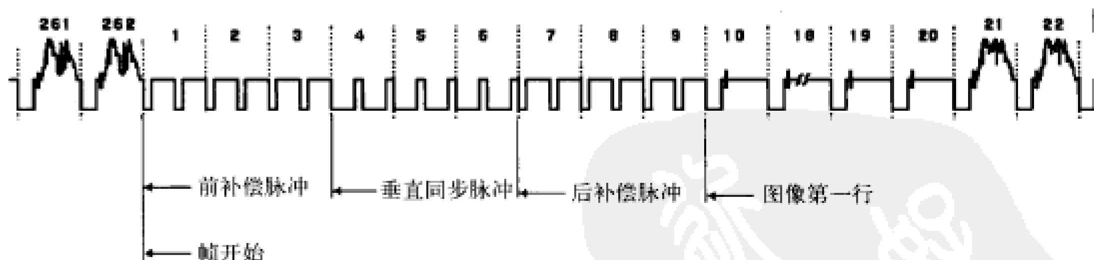


图 12-3 完整的视频帧信号

现在要注意，除了组成每一帧的所有行外，还有 3 个行周期的延长同步脉冲用来提供垂直同步信息，表明每一帧的开始。它们的前面和后面都有 3 个附加行，分别是前补偿和后补偿行。

12.2.1 产生合成视频信号

如果要将项目限制为只生成简单的黑白图像（没有灰度，没有色彩）和一个非逐行扫描图像，那么所需的硬件和软件将大大减少，特别是硬件接口只需要 3 个合适阻值的电阻连接到两个数字 I/O 引脚（如图 12-4 所示）。其中一个 I/O 引脚会产生同步脉冲，而另一个 I/O 引脚会产

生实际的亮度信号。

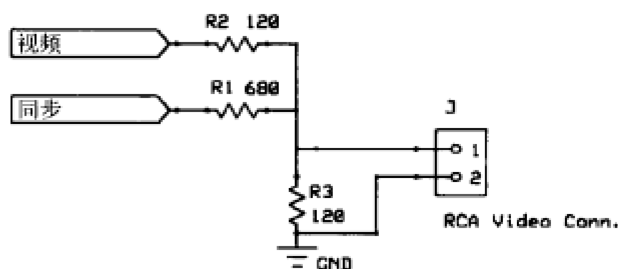


图 12-4 NTSC 视频输出的简单硬件接口

三个电阻的值必须选择恰当，这样亮度和同步信号的相对值才能接近标准的 NTSC 规范(如表 12-2 所示)，而信号的总幅值接近 1 V 的峰峰值，电路的输出阻抗接近 75 Ω。按照前一幅图上的电阻值，就可以产生黑白图像的 3 个基本信号电平，如图 12-5 所示。

表 12-2 产生亮度和同步脉冲

信号特征	同步	视频
同步脉冲	0	0
黑行	1	0
白行	1	1

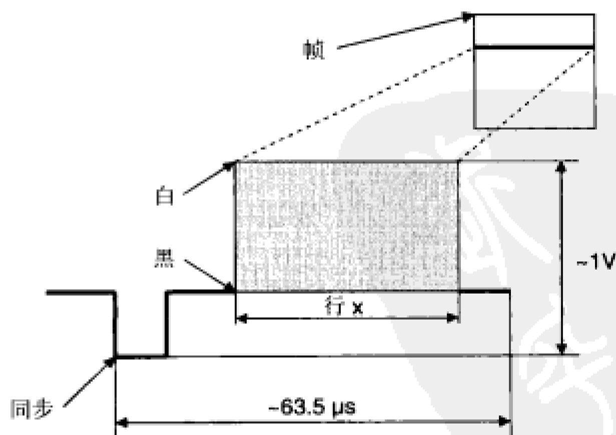


图 12-5 简化的 NTSC 合成信号

由于这里不打算使用隔行扫描，因此可以使每个周期只产生垂直同步脉冲，以简化前补偿、垂直同步和后补偿脉冲，如图 12-6 所示。

现在，产生完整视频输出信号的问题，就可以简化成一个状态机，使用一个定时器中断在固定周期内触发就可以了，如图 12-7 所示。状态机是很简单的，每个状态对应帧的一个行类型，

在转换到下一个状态前，重复固定的次数就可以了。

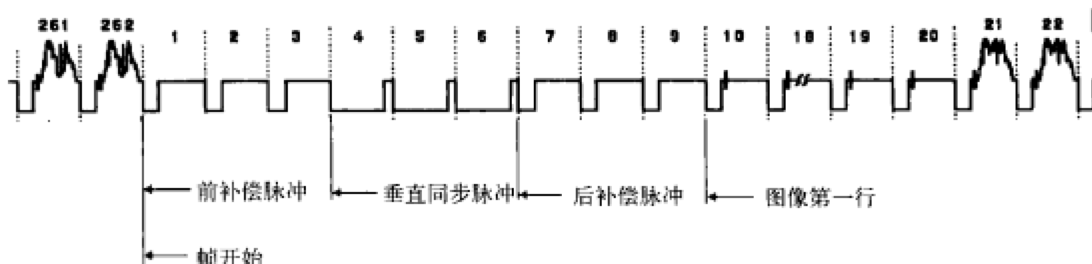


图 12-6　简化的 NTSC 视频帧（非隔行扫描）

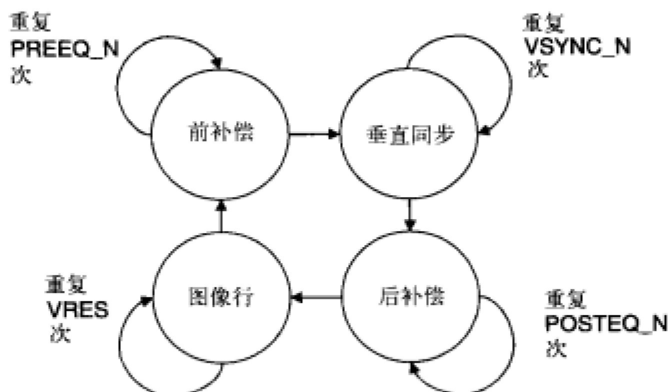


图 12-7　视频状态机示意图

一个简单的表就能说明每个状态的转换，见表 12-3。

表 12-3　视频状态机转换表

状　态	重　复	转 换 到
前补偿	PREEQ_N 次	垂直同步
垂直同步	3 次	后补偿
后补偿	POSTEQ_N 次	图像行
图像行	VRES 次	前补偿

垂直同步行的数量是固定的，并且在 NTSC 视频标准中早有规定，而每一帧中所包含图像的实际行数是由用户自定义的（当然有规定的范围）。尽管理论上可以用尽显示屏上所有的行来显示最大量的数据，然而还要考虑实际的一些限制，特别是 PIC24FJ128GA010 上可用于视频图像的 RAM 存储空间。这些限制规定了用于显示图像的行数（VRES），而剩下的行数（多至 NTSC 标准行数）则显示空白。

实际上，如果 V_NTSC 表示标准 NTSC 视频帧的总行数，VRES 表示期望的垂直分辨率，那么 PREEQ_N 和 POSTEQ_N 就可以定义为：

```
#define V_NTSC    262      // total number of lines composing a frame
#define VSYNC_N   3        // V sync lines

// count the number of remaining black lines top+bottom
#define VBLANK_N     (V_NTSC -VRES - VSYNC_N)

#define PREEQ_N    VBLANK_N /2           // pre equalization + bottom blank lines
#define POSTEQ_N   VBLANK_N - PREEQ_N    // post equalization + top blank lines
```

如果选择 Timer3 作为时基，那么可以将它的周期寄存器 PR3 初始为在指定周期产生中断，而在它的中断子程序中可以插入上面的状态机。下面是完成视频发生逻辑的中断服务子程序的框架：

```
// next state table
int VS[4] = { SV_SYNC, SV_POSTEQ, SV_LINE, SV_PREEQ};
// next counter table
int VC[4] = { VSYNC_N,  POSTEQ_N,    VRES,  PREEQ_N};


void _ISRFAST _T3Interrupt( void)
{
    // Start a Sync pulse
    SYNC = 0;

    // decrement the vertical counter
    VCount--;

    // vertical state machine
    switch ( VState) {
        case SV_PREEQ:
            // horizontal sync pulse
            ...
            break;

        case SV_SYNC:
            // vertical sync pulse
            ...
            break;

        case SV_POSTEQ:
            // horizontal sync pulse
            ...
            break;

        default:
        case SV_LINE:
        ...
    } //switch

    // advance the state machine
    if ( VCount == 0)
    {
        VCount = VC[ VState];
        VState = VS[ VState];
    }
```

```
    // clear the interrupt flag
    _T3IF = 0;

} // T3Interrupt
```

一旦进入中断服务子程序，就可以立即将同步输出引脚电压降低，来产生水平同步脉冲，不过还需要另一个机制来提供正确的定时（大约是 4.5 μs）以完成脉冲（上升沿），并产生剩下的水平行波形。这里有以下几种方法值得研究。

(1) 使用定时器产生短延时循环。

(2) 使用另一个定时器，配合中断服务子程序。

(3) 使用输出比较模块，配合中断服务子程序。

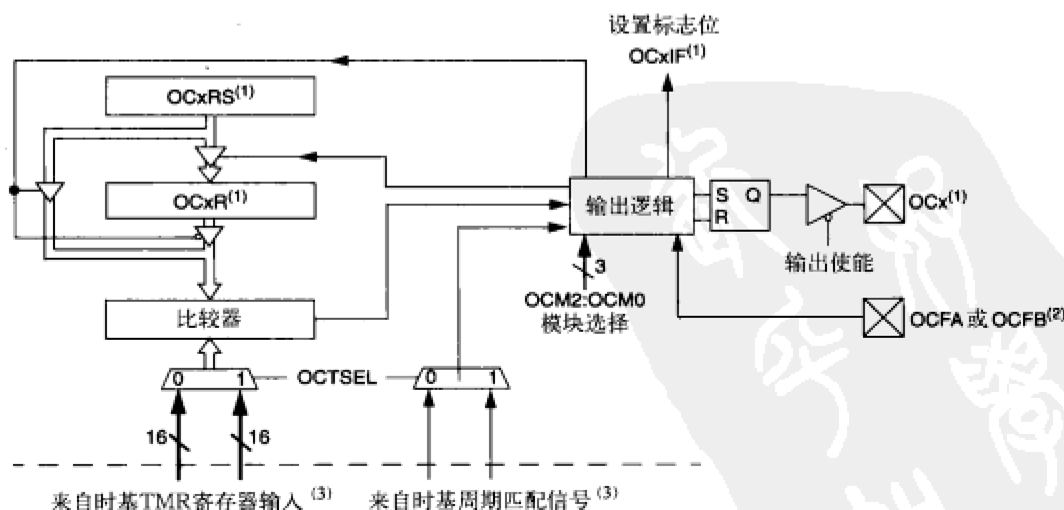
第一种方法的程序编码最简单，不过明显的缺点是浪费了大量的处理器周期（4.5 微秒 × 16 个周期每微秒=72 个周期），也就是重复的每个水平行周期（63.5 μs 或者大约 1 018 个周期）会占用大约 7%的处理器资源。

第二个方法显然更有效，而且到目前为止，读者已经积累了很多的经验，可以使用定时器中断和它们的中断服务子程序来实现小型的状态机。

第三种方法需要用到一种在前面章节中还没有介绍过的新外设，因此需要更多的关注。

12.2.2　使用输出比较模块

PIC24FJ128GA010 微控制器有 5 个输出比较模块，可以用于多种用途，包括生成单脉冲、生成连续脉冲，以及脉冲宽度调制（PWM）。每个模块都可以连接到两个 16 位定时器的其中一个（Timer2 或者 Timer3），在必要时，它的一个输出引脚还可以用于触发和产生上升或者下降沿。最重要的是，每个模块都有一个关联的独立中断向量（如图 12-8 所示）。



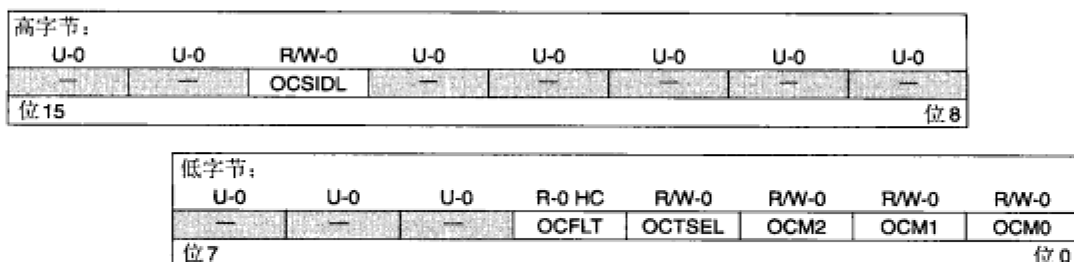
(1) “x” 表示对应的输出比较通道，数值从 1 到 8。

(2) OCFA 引脚控制 OC1~OC4 通道。OCFB 引脚控制 OC5~OC8 通道。

(3) 每个输出比较通道可以选择两种时间基中的一种。请参阅关于模块的时间基的设备数据表。

图 12-8　输出比较模块图

在单脉冲模式下，OCxR 寄存器可用于确定中断事件的触发时间（与所选定时器的值有关），并且如果有需要，输出引脚会被设置/重设或者触发，如图 12-9 所示。



高字节：

U-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
—	—	OCSIDL	—	—	—	—	—
位 15							位 8

低字节：

U-0	U-0	U-0	R-0 HC	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	OCFLT	OCTSEL	OCM2	OCM1	OCM0
位 7							位 0

图 12-9 输出比较控制寄存器 OCxCON

OCxCON 寄存器是用来控制每个输出比较模块的唯一配置寄存器。

在这里，输出比较机制是非常有用的，因为有两个地方需要精确的定时：水平同步脉冲的终止，用于产生前/后补偿或者垂直同步行；还有就是后沿的终止，用于开始真实的图像显示，如图 12-10 所示。

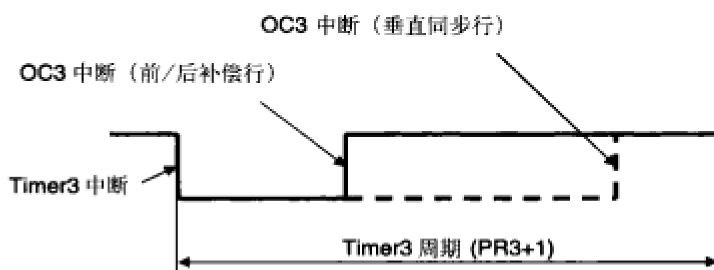


图 12-10 同步行的中断序列

选择其中一个输出比较模块（我们选择 OC3）来确定同步行的精确结束时间。这里不需要连接输出引脚（RD2），而是在对应的中断服务子程序里提高同步信号。

```
void _ISRFAST _OC3Interrupt( void)
{
    SYNC = 1;    // bring the output up to the black level
    _OC3IF = 0; // clear the interrupt flag
} // OC3Interrupt
```

在单脉冲模式下（OCM=001），OC3CON 控制寄存器会置 1，以触发输出比较模块，并且使用 Timer3 作为参考时间基（OCTSEL=1）。

根据行的类型（状态机状态），使用选择的定时值对 OC3R 寄存器进行初始化：

```
    // vertical state machine
    switch ( VState) {
        case SV_PREEQ:
            // horizontal sync pulse
             OC3R = HSYNC_T;
             OC3CON = 0x0009;     // single event mode
```

```
            break;

        case SV_SYNC:
            // vertical sync pulse
            OC3R = H_NTSC - HSYNC_T;
            OC3CON = 0x0009;     // single event mode
            break;

        case SV_POSTEQ:
            // horizontal sync pulse
            OC3R = HSYNC_T;
            OC3CON = 0x0009;     // single event mode
...
```

在产生视频行时，使用另一个输出比较模块（OC4）来标志后沿的结束，相应的中断服务子程序将被用来初始化图像行的视频流，如图 12-11 所示。

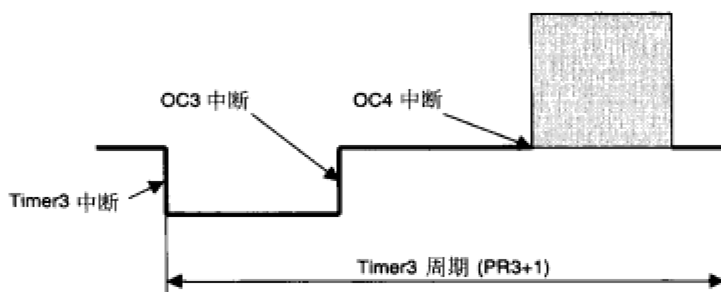


图 12-11 视频行的中断序列

```
case SV_LINE:
    // activate OC3 for the end of the horizontal sync pulse
    OC3R = HSYNC_T;
    OC3CON = 0x0009;     // single event

    // activate OC4 for the end of the back porch
    OC4R = HSYNC_T + BPORCH_T;
    OC4CON = 0x0009;     // single event
    ...
    break;
```

12.2.3 存储器分配

到目前为止，讨论的都是组成 NTSC 视频波形的同步信号的生成，由简单硬件接口的两个 I/O 中的一个控制。另一个 I/O 则在产生包含真正图像的行时会用到。转换视频的 I/O，就可以让行的片段显示为白（0）或者黑（1）。由于 NTSC 标准规定了亮度信号的最大带宽为 4.2MHz，而前沿和后沿的间隔为 52 μs 宽，因此可以显示的黑白片段最多是 218（52×4.2），换而言之，理论上，每行的水平像素值是 436（假设整个屏幕两边都用上）。最大的垂直分辨率由 NTSC 标准的最大行数减去补偿和垂直同步的最少行数得到 253。如果要生成最大可能的图像，那应该是由 253×436 像素矩阵，即 110 308 个像素点组成。更进一步地，如果每个像素用 1 位表示，

那么就需要分配 13.5 KB 的数组，对于 PIC24FJ128GA010 上只有 8 KB 的 RAM 空间来说，实在是太大了。实际上，要保证产生高分辨率的输出，同时还要让图像能塞进 RAM 存储器中，并且留有足够的空间来运行程序，以为栈和变量腾出空间。当然，有很多方法可用来将水平分辨率值和垂直分辨率值组合成可接受的存储空间，这里考虑两点来计算合适的数量：假设使用的是整数数组，那么将水平值定为 16 的倍数，可以用数学方法更快地找出像素在存储区的位置。同样，将两个数值的比值设在 4∶3 附近，可以避免几何畸变（换而言之，画出的圆看起来更像是圆形而不是椭圆）。

将水平分辨率定为 256 个像素（HRES），垂直分辨率定为 192 个像素，那么一幅图像所需要的存储空间就是 6 144 字节（256×192/8），还有 2 048 字节的空间留给了栈和变量。

通过 C30 编译器，可以很容易地分配 1 个整数数组（每字每次有 16 个像素）给整个图像的变址。不过还需要确保数组是可寻址的，并且不能简单定义成 near 变量（使用小存储模块时的默认值）。near 变量必须位于数据地址的前 8 KB 位置，不过该位置页包含了特殊功能寄存器和 PSV 区。最能有效避免出现分配错误信息的方法，就是将视频存储变址定义成 far 属性：

```
#define _FAR __attribute__(( far))

int _FAR VMap[VRES * (HRES/16)];
```

这样就可以保证数组的元素是通过指针访问的，包括读和写。

12.2.4 图像串行化

如果每个图像行是以存储器中 VMap 数组的一行 16 个整数表示的，那么就需要在合成视频波形的后沿和前沿中间短短的时间（52 μs）内，连续地输出每一位（像素）。

换而言之，每 200 ns 或者更短的时间内，需要对选定的视频输出引脚设置或者复位一个新的像素值。然后转换成像素间的 3 个状态机周期，这样短的时间，对于一个简单的转换循环，甚至是用汇编语言直接解码都是不可能的。更麻烦的是，即使在如此紧迫的时间里可以完成一个循环，可是这也会占用视频生成的大部分处理时间，给主程序留下的只有很少的处理器周期（最多只有 18%）。幸好，PIC24 有一个外设可以有效地解决图像串行化问题，那就是 SPI 同步串行通信模块。

在前面的章节中，曾经使用 SPI2 端口来同串行 EEPROM 存储器进行通信。当时，读者应该注意到 SPI 模块实际上是由外部时钟信号（从模式）或者内部时钟（主模式）控制的一个移位寄存器组成的。在新的项目中，可以将 SPI1 模块用于主模式，直接连接 SDO（串行数据输出）和硬件接口的视频引脚，而 SDI（数据输入）和 SCK（时钟输出）、SS（从选择）引脚保持空置。在 PIC24 SPI 模块的众多新的高级功能中，有两个功能特别适合视频应用：16 位工作模式和 8 层 FIFO 缓冲器。16 位的工作模式，可以让图像存储变址映像与 SPI 模块之间的数据传输速度翻倍。而 8 层的 FIFO 缓冲器可以一次最多向 SPI 缓冲器装载 128 个像素，并且迅速地返回中断服务子程序，只需经过 25 μs 就又可以进行下一次装载，仅需要每个图像行的两个短脉冲就可以获得最大化的视频发生器效率。

下面开始编写第二个输出比较模块的中断服务子程序，配置为在后沿结束后立即触发状态机，产生图像的行输出：

```
void _ISRFAST _OC4Interrupt( void)
{
    // load SPI FIFO with 8 x 16-bit words = 128 pixels
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;

    if ( --HCount > 0)
    { // activate again in time for the next SPI load
        OC4R += ( PIX_T * 7 * 16);
        OC4CON = 0x0009;    // single event
    }

    // clear the interrupt flag
    _OC4IF = 0;

} // OC4Interrupt
```

注意中断服务子程序在向 SPI 缓冲器装载第一批 128 个像素数据后，如何为第二个脉冲（第二个半图像行）重新设置 OC4 模块的。

现在，已经对各个模块进行了定义，下面要编写视频发生器所有模块的初始化子程序：

```
void initVideo( void)
{
    // set the priority levels
       _T3IP = 4;            // this is the default value anyway
       _OC3IP = 4;
       _OC4IP = 4;

       TMR3 = 0;             // clear the timer
       PR3 = H_NTSC; // set the period register to NTSC line

    // 2.1 configure Timer3 modules
       T3CON = 0x8000;       // enabled, prescaler 1:1, internal clock

    // 2.2 init Timer3/OC3/OC4 Interrupts, clear the flag
       _OC3IF = 0;    _OC3IE = 1;
       _OC4IF = 0; _OC4IE = 1;
       _T3IF = 0;   _T3IE = 1;

    // 2.3 init the processor priority level
       _IP = 0;              // this is the default value anyway

    // init the SPI1
    if ( PIX_T == 2)
        SPI1CON1 = 0x043B;  // Master, 16 bit, disable SCK/SS, prescale 1:3

    else
        SPI1CON1 = 0x0437;  // Master, 16 bit, disable SCK/SS, prescale 1:2
```

```
    SPI1CON2 = 0x0001;        // Enhanced mode, 8 x FIFO
    SPI1STAT = 0x8000;        // enable SPI port

    // init PORTF for the Sync
    _TRISG0 = 0;              // output the SYNC pin

    // init the vertical sync state machine
    VState = SV_PREEQ;
    VCount = PREEQ_N;

} // initVideo
```

注意，参数 PIX_T 可以用于选择不同的 SPI 时钟预分频器值，以适应不同的水平分辨率的需要。当 PIX_T=3 时，可为每个像素提供 3 个时钟周期（总共 187.5 ns），非常接近于过去对 256 像素水平分辨率计算出的 200 ns，这将获得最小的图像损失。

12.2.5 构建视频模块

现在，加入所有的定义和必要的引脚分配，就可以完成整个视频状态机的编码了：

```
/*
** NTSC Video using T3 and Output Compare interrupts
**
*/

#include <p24fj128ga010.h>
#include "Graphic.h"

// I/O definitions
#define SYNC      _LATG0   // output
#define SDO       _RF8     // SPI1 SDO

// timing definitions for NTSC video vertical state machine
#define V_NTSC    262      // total number of lines composing a frame
#define VSYNC_N   3        // V sync lines

// count the number of remaining black lines top+bottom
#define VBLANK_N     (V_NTSC -VRES - VSYNC_N)

#define PREEQ_N    VBLANK_N /2           // pre equalization + bottom blank lines
#define POSTEQ_N   VBLANK_N - PREEQ_N    // post equalization + top blank lines

// definition of the vertical sync state machine
#define SV_PREEQ     0
#define SV_SYNC      1
#define SV_POSTEQ    2
#define SV_LINE      3

// timing definitions for NTSC video horizontal state machine
#define H_NTSC   1018     // total number of Tcy in a line (63.5us)
#define HSYNC_T   90      // Tcy in a horizontal sync pulse
#define BPORCH_T 90       // Tcy in a back porch
#define PIX_T     3       // Tcy in each pixel, valid values are only 2 or 3
```

```
#define _FAR __attribute__(( far))

int _FAR VMap[VRES * (HRES/16)];

volatile int *VPtr;
volatile int HCount, VCount, VState, HState;

// next state table
int VS[4] = { SV_SYNC, SV_POSTEQ, SV_LINE, SV_PREEQ};
// next counter table
int VC[4] = { VSYNC_N,  POSTEQ_N,    VRES,  PREEQ_N};


void _ISRFAST _T3Interrupt( void)
{
    // Start a Sync pulse
    SYNC = 0;

    // decrement the vertical counter
    VCount--;

    // vertical state machine
    switch ( VState) {
        case SV_PREEQ:
           // horizontal sync pulse
            OC3R = HSYNC_T;
            OC3CON = 0x0009;    // single event
            break;

        case SV_SYNC:
            // vertical sync pulse
            OC3R = H_NTSC - HSYNC_T;
            OC3CON = 0x0009;    // single event
            break;

        case SV_POSTEQ:
            // horizontal sync pulse
            OC3R = HSYNC_T;
            OC3CON = 0x0009;    // single event
            // on the last posteq prepare for the new frame
            if ( VCount == 0)
            {
                VPtr = VMap;
            }
            break;

        default:
        case SV_LINE:
            // horizontal sync pulse
            OC3R = HSYNC_T;
            OC3CON = 0x0009;    // single event

            // activate OC4 for the SPI loading
            OC4R = HSYNC_T + BPORCH_T;
```

```
            OC4CON = 0x0009;    // single event
            HCount = HRES/128;  // loads 8x16 bits at a time
            break;
    } //switch

    // advance the state machine
    if ( VCount == 0)
    {
        VCount = VC[ VState];
        VState = VS[ VState];
     }

    // clear the interrupt flag
    _T3IF = 0;

} // T3Interrupt
```

要得到完整的库模块，需要加入本章前面已经介绍过的输出比较模块 OC3 和 OC4 的中断服务子程序，以及一些额外的附加函数：

```
void clearScreen( void)
{
    int i, j;
    int *v;

    v = (int *)&VMap[0];

    // clear the screen
    for ( i=0; i < (VRES*( HRES/16)); i++)
        *v++ = 0;
} //clearScreen


void haltVideo( void)
{
    T3CONbits.TON = 0;   // turn off the vertical state machine
} //haltVideo

void synchV( void)
{
    while ( VCount != 1);
} // synchV
```

特别是 clearScreen 函数对于初始化图像的存储器映射和 VMap 数组是非常有用的，而 haltVideo 函数可以有效地中断视频的发生，让 PIC24 处理器以 100%的能力处理其他重要任务/应用。

synchV 函数用于视频发生器与任务的同步，这个函数只能是在视频发生器开始"描绘"屏幕的最后一行时返回。对于图形显示，它可以最小化闪烁并且/或者提供更流畅的滚动和动画。

将以上所有函数保存在文件"graphic.c"中，并加入到新项目"video"中。

然后生成一个新文件，并加入下面的定义：

```
/*
**   NTSC Video
**   Graphic library
**
*/

#define VRES      192    // desired vertical resolution
#define HRES      256    // desired horizontal resolution (pixel)

void initVideo( void);

void haltVideo( void);

void clearScreen( void);

void synchV( void);

extern int VMap[HRES/16*VRES];
```

将上面的文件保存为“graphic.h”，添加到相同的项目中。

注意，水平分辨率和垂直分辨率的值只是两个公开的参数。在一些合理的限制（定时约束以及很多前面提到的考虑因素）范围内，这两个值可以根据特定的需要进行改变，因此，视频发生器模块的状态机和其他机制都将根据需要调整他们的时序。

12.2.6 视频发生器测试

为了测试刚刚完成的视频发生器模块，只需用到 MPLAB SIM 仿真器工具和必要的一些主程序代码行：

```
//
// Graphic Test.c
//
// testing the basic graphic module
//

#include <p24fj128ga010.h>
#include "../graphic/graphic.h"

main()
{
    // initializations
    TRISA = 0xff80;       // set PORTA lsb as output for debugging
    clearScreen();        // init the video map
    initVideo();          // start the video state machine

    // main loop
    while( 1)
    {

    } // main loop

} // main
```

保存项目，并使用项目构建列表构建完整的项目。

打开逻辑分析器窗口，使用逻辑分析器列表添加 RG0 引脚（sync）和 SDO1 输出（视频）到分析器通道。此时就可以仿真运行几秒钟，然后按下暂停，转到逻辑分析器输出窗口来观察结果，如图 12-12 所示。仿真器的追踪存储的能力有限，只可以观察到整个视频帧的一小部分。换而言之，能看到的只是相对无趣的显示，只有一些常规的 sync 脉冲序列和平常的视频输出。然而，仿真器并不能仿真 SPI 端口的输出，因此只能使用真实的硬件来运行程序才能看到输出。对于每一个 sync 信号，有一个有趣的时刻是值得关注的——在每一帧开始处，产生带有 3 个宽的水平同步脉冲的垂直同步信号的时候。在 OC4 中断服务子程序的第一行处设置断点（第一次调用图像第一行的开始），可以确定仿真器将停止在新一帧开始前的地方。

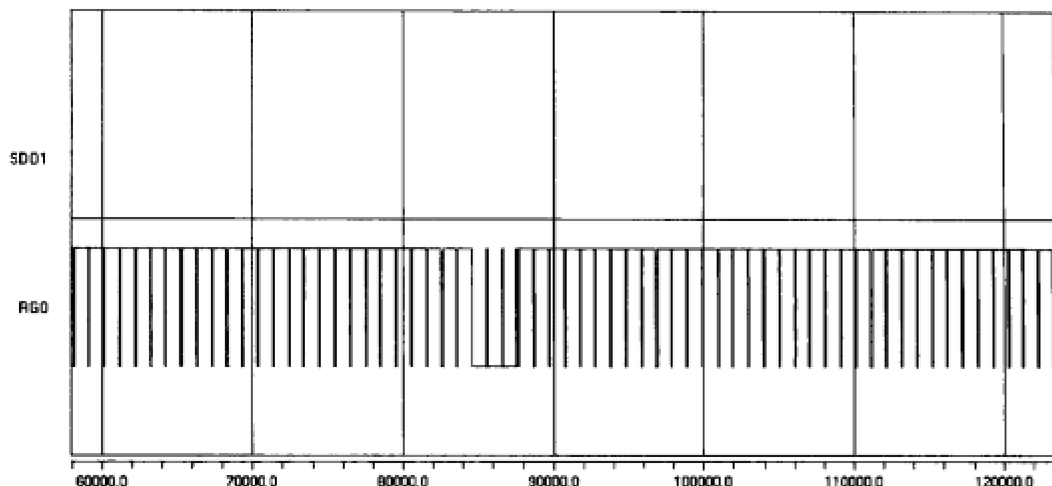


图 12-12 逻辑分析器窗口屏幕截图（垂直 sync 脉冲）

如果读者有耐性的话，数数后面 3 个垂直 sync（长）脉冲的行数，那么可以看出那是 33 行（即(262−192−3)/2）。同时，读者也可以放大图像显示验证出现在前/后补偿和垂直脉冲线的 sync 脉冲时序，如图 12-13 所示。

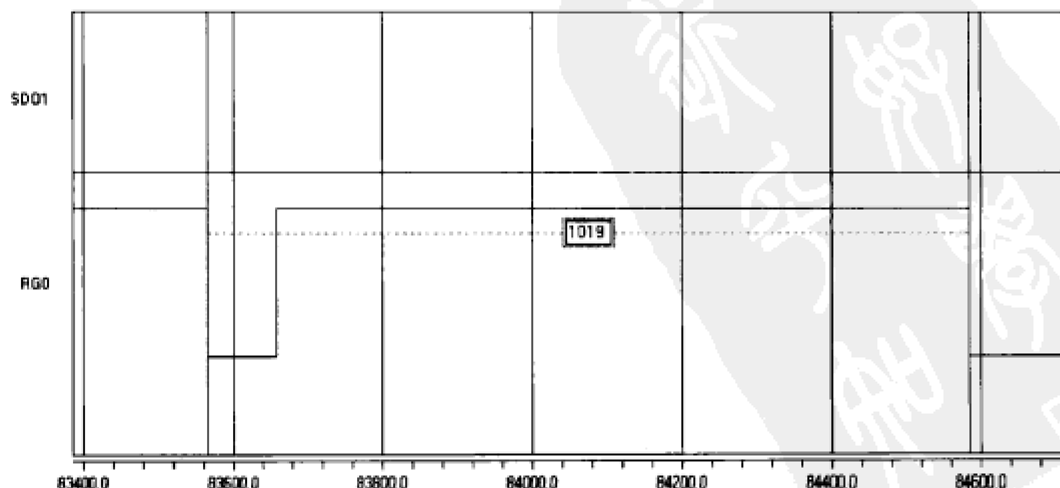


图 12-13 单个前补偿行的放大图

使用光标，可以验证组成水平行的周期数量以及水平 sync 脉冲宽度。记住逻辑分析器窗口的读数近似于最近的屏幕像素，因此读数的精度取决于放大倍数和 PC 屏幕的分辨率。如果读者只需要确定精确的时间间隔，那么最直接的方法就是使用 MPLAB SIM 软件仿真器的 stopwatch 功能以及适当的断点设置。

12.2.7 性能测定

由于视频发生器使用 3 种不同的中断源和一个带 4 种状态的状态机，因此真实处理器的有关管理是很有趣的，可以使用逻辑分析器揭示处理器在每种中断服务子程序上的时间开销比例，如图 12-14 所示。

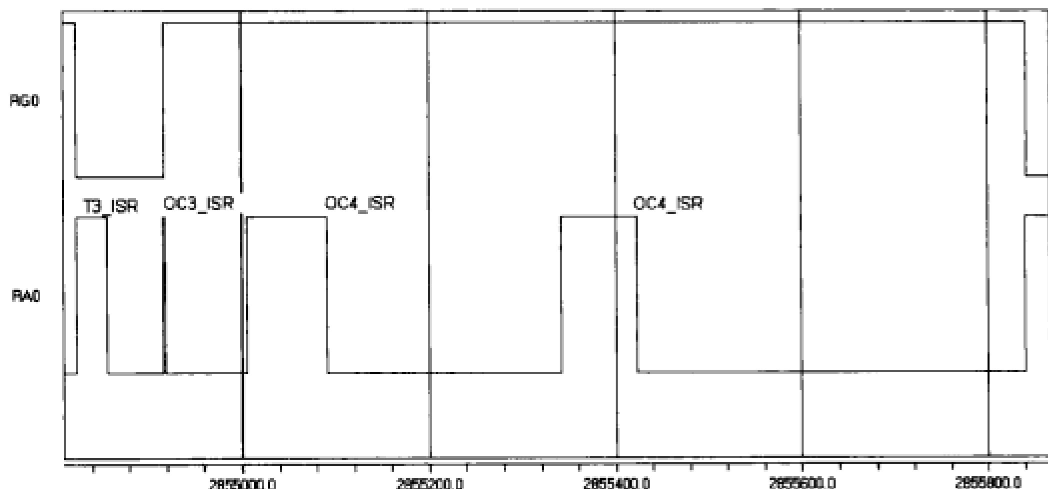


图 12-14 逻辑分析器输出的屏幕截图（测试性能）

最后，需要对 3 种中断服务子程序作一些简单的修改。使用 PORTA（RA0）的其中一个引脚作为标志位，表示正在执行中断子程序，在执行主程序时清零：

```
void _ISRFAST _T3Interrupt( void)
{
    _RA0=1;
...
     _RA0=0;
} // T3Interrupt

void _ISRFAST _OC3Interrupt( void)
{
     _RA0=1;
...
     _RA0=0;
} // OC3Interrupt

void _ISRFAST _OC4Interrupt( void)
{
     _RA0=1;
...
     _RA0=0;
} // OC4Interrupt
```

经过了重新编译以及将 RA0 添加到逻辑分析器的输入捕捉，可以放大进入单个的水平行周期（选择一个图像行）。

使用光标，可以计算出每个中断服务子程序大概的持续时间，通过求和得到可能的最坏值（4 个中断都用到的图像行），这里得出的是一行 1 018 周期中的 200 个周期，表示占用的处理器时间少于 20%，这是一个不错的值。

12.2.8 暗屏

使用仿真器和逻辑分析器工具读者会得到一定的乐趣，不过相信读者忍不住要感受一下真实的器材了！读者可以在真实的 TV 屏幕（或者其他能接收 NTSC 合成视频信号的设备）上测试视频接口，只需连接真实 PIC24 设备上简单的三电阻接口。如果读者有 Explorer16 板，现在就拿出焊铁，将 3 个电阻焊接到演示板右上角的原型区和标注的 RCA 视频插座。读者觉得自己对电子爱好已超出该任务，可以开发小的 PCB 作为 Explorer16 的扩展子板。

> 登陆配套的网站 www.flyingthepic24.com，查找适用的扩展板，读者会找到适于本书第三部分全部进阶项目的内容。

无论读者选择哪一个，实验都将是激动人心的。

或者不是！实际上，如果读者在 Explorer16 演示板通电时就做好所有的连接，看到的将会是空白（或者更准确地说是全黑）的屏幕。当然，实验成功了。实际上，这已经表明很多的工作已经正确，水平和垂直同步信号已经由 TV 正确地解码，一个漂亮的全黑的背景已经显示出来，如图 12-15 所示。



图 12-15 暗屏

12.2.9 测试图样

为了让测试工作更加有趣，应该添加一些有观赏性的视频数组，即一些简单的，可以立即给出视频发生器功能的反馈信息。

下面生成如下的新的测试程序：

```
//
// Graphic Test2.c
//
// testing the basic graphic module
//

#include <p24fj128ga010.h>
#include "../graphic/graphic.h"

main()
{
    int x, y;

    // fill the video memory map with a pattern
    for( y=0; y<VRES; y++)
        for (x=0; x<HRES/16; x++)
            VMap[y*16 + x]= y;

    initVideo();     // start the video state machine

    // main loop
    while( 1)
    {

    } // main loop

} // main
```

这次使用两个嵌套的 for 循环来初始化 VMap 映射，而不是调用 clearScreen 清屏函数。外层的（*y*）循环用于垂直行，内部的（*x*）循环用于水平移动，将 16 个字（每个有 16 位）填充相同的值：行号。换而言之，第一行的每个字的值都是 0，第二行每个字的值都是 1，如此类推直到最后一行（192）的每个字的值是 191（十六进制的 0xBF）。

构建新项目测试视频输出，将会看到如图 12-16 所示的图样。

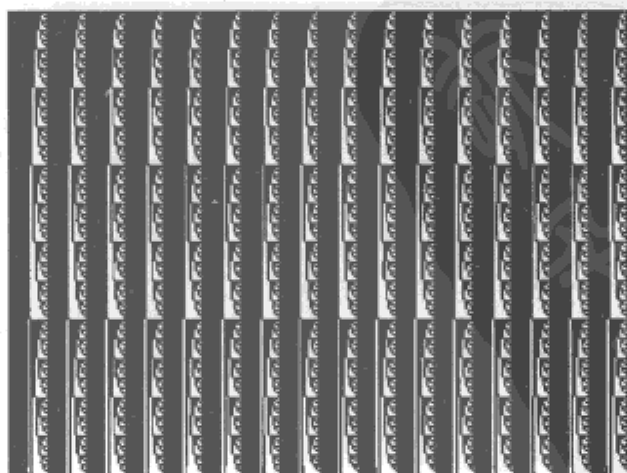


图 12-16　使用测试图样产生的视频输出截图

简单说来，有很多知识可以从观察上面的测试图样中学到。首先，注意每个字以二进制显示在屏幕上时的最高位都是在左边。这是由 SPI 模块的移位方式决定的：实际上是 msb 优先。

第二，可以验证最后一行的图样是期望的 0x00bf，因此可以知道存储映射的每一行都显示了。第三，还可以欣赏一下图像的细节。不同的输出设备（TV、投影仪、LCD 板等）或多或少可以锁定图像，和/或根据本身的显示分辨率和输出带宽锐化图像。总的来说，读者可以看到 PIC24 可以生成垂直的竖线。这并非微不足道。其实，每个像素一行一行地对齐成笔直的竖线，需要绝对精确地响应每个定时器中断，这也是 PIC®微控制器结构的值得称道的特性。

但是这并不意味着在最大的屏幕上，完全没有任何的瑕疵，因为输出图像里会有微小的回显和视觉假象。实际上，简单的三电阻接口能达到的质量就只有这个程度了。

最终，整个的合成视频信号接口的输出质量也不好。正如读者知道的，S-视频、VGA 和其他大多数的视频接口是将亮度和同步信号分开处理以获得更稳定和清晰的图像的。

12.2.10 描点

现在，需要再次确认图形显示模块的功能，把更多的注意力投入到存储映射的图样生成上。首先第一步自然是要生成一个函数，可以点亮屏幕上精确坐标(x, y)上的像素。第一件要做的事情是从 y 坐标提取行号。如果 x 和 y 坐标采用传统的直角坐标系，而原点位于屏幕的左下角，那么就需要在访问存储映射前插入地址，因此存储映射的第一行对应 y 的最大坐标是 `VRE-1` 或者 189，而存储映射的最后一行对应的 y 坐标是 0。同样，因为存储映射的行有 16 个字，因此需要将每一行的号码乘以 16 来得到对应行的第一个字的地址。可以使用下面的表达式概括为：`VMap[ (VRES-1-y)*16]`。

因为像素以 16 位字分组，所以要解决 x 坐标的问题，首先需要确定目标像素的字。简单地除以 16 就可以得到字偏移量。将字偏移量加上前面计算的行地址，就可以得到存储映射中完整的字地址：

```
VMap[ (VRES-1 -y) *16 + (x/16)]
```

为了优化地址计算，可以使用移位操作来执行乘法和除法运算：

```
VMap[ (VRES-1 -y) << 4 + (x>>4)]
```

要识别目标像素对应的字位的位置，可以使用 x 除以 16 所得的余数，或者更有效的方法是屏蔽 x 坐标的低四位。因为要点亮像素，需要使用合理的屏蔽执行二进制 OR 操作，即屏蔽对应的像素位置的位。要记住，显示将每个字的 msb 放在左边（SPI 模块先移位 msb），可以使用以下的语句构建屏蔽：

```
( 0x8000 >> ( x & 0xf))
```

将所有的语句放在一起，就得到描点的核心函数：

```
VMap[ ((VRES-1-y)<<4) + (x>>4)] |= (0x8000 >> (x & 0xf));
```

最后，可以添加“clipping”（剪切），这是一个简单安全的检查，保证所分配的坐标是有效的，并且位于当前屏幕映射的范围内：

```
void Plot( unsigned x, unsigned y)
{
    if ((x<HRES) && (y<VRES) )
        VMap[ ((VRES-1-y)<<4) + (x>>4)] |= (0x8000 >> (x & 0xf));
} // plot
```

通过将 x 和 y 参数定义成 unsigned 整数，可以避免出现负值，因为它们是被认作超出屏幕的范围的。

12.2.11　星夜

为了测试新建的描点函数，需要生成一个新的项目。这将会用到“graphic.c”和“graphic.h”文件，同时还会用到标准 C 库“stdlib.h”中的伪随机数发生函数。通过伪随机数发生器产生 1 000 个 x 和 y 坐标，就可以使用下面的简单代码同时测试描点函数和随机发生器：

```
//
// Graphic Test3.c
//
// testing the basic graphic module
// plotting random points
//

#include <p24fj128ga010.h>

#include "../graphic/graphic.h"
#include <stdlib.h>


void plot( unsigned x, unsigned y)
{
      if ((x<HRES) && (y<VRES) )
         VMap[ ((VRES-1-y)<<4) + (x>>4)] |= (0x8000 >> (x & 0xf));
} // plot


main()
{
    int i;

    // initializations
    clearScreen();        // init the video map
    initVideo();          // start the video state machine

    srand(13);            // initialize the pseudo random number generator

    for( i=0; i<1000; i++)
    {
        plot( rand()%HRES, rand()%VRES);
    }

    // main loop
    while( 1)
    {

    } // main loop

} // main
```

视频显示的输出看起来就像是满布星星的夜空，如图 12-17 所示的屏幕截图。



图 12-17 屏幕截图（描绘星夜）

读者可能会注意到，这个星夜并不是很真实，因为没有看到一个可以察觉到的、密度明显较高的带——换而言之，没有银河！这实际上是好事！它说明了伪随机数生成器工作正常。

现在可以将描点函数加入到 “graphic.c” 模块。要记住还要添加到 “graphic.h” 函数，因为在下面的练习中还会用到它：

```
void plot( unsigned, unsigned);
```

12.2.12 画线

接下来要做的显然就是画线，或者更准确地说是线段。诚然，要画出水平和垂直的线段都不是什么问题了。简单的 for 循环就可以搞定，不过画斜线完全是另一码事。或者可以从初中时期的两点一线方程开始：

```
y = y0 + (y1-y0)/(x1-x0) * ( x- x0)
```

其中 (x0,y0) 和 (x1,y1) 分别是线段两个端点的坐标。

这个方程给出的含义是，对于任何给定的 x 坐标，则有对应的 y 坐标。对于线段之间的离散坐标值 x，就在下面的循环中使用该方程：

```
//
// Line Test1.c
//
// testing the basic line drawing function
//

#include <p24fj128ga010.h>

#include "../graphic/graphic.h"

main()
{
```

```
    int x;
    float x0 = 10, y0 = 20, x1 = 200, y1 = 150, x2 = 20, y2 = 150;

    // initializations
    clearScreen();  // init the video map
    initVideo();    // start the video state machine

    // draw an oblique line (x0,y0) - (x1,y1)
    for( x=x0; x<x1; x++)
        plot( x, y0+(y1-y0)/(x1-x0)* (x-x0));

    // draw a second (steeper) line (x0,y0) - ( x2,y2)
    for( x=x0; x<x2; x++)
        plot( x, y0+(y2-y0)/(x2-x0)* (x-x0));


    // main loop
    while( 1)
    {

    } // main loop

} // main
```

输出产生的第一条（较平缓的）线段比较容易接受，它的水平距离 x1–x0 要大于垂直距离 y1–y0。第二条更陡峭的线段上的点太分散了，是不能接受的结果。（如图 12-18 所示。）同样，可以使用浮点机制，但与整数机制相比，计算量开销比较大，正如前面章节所看到的。

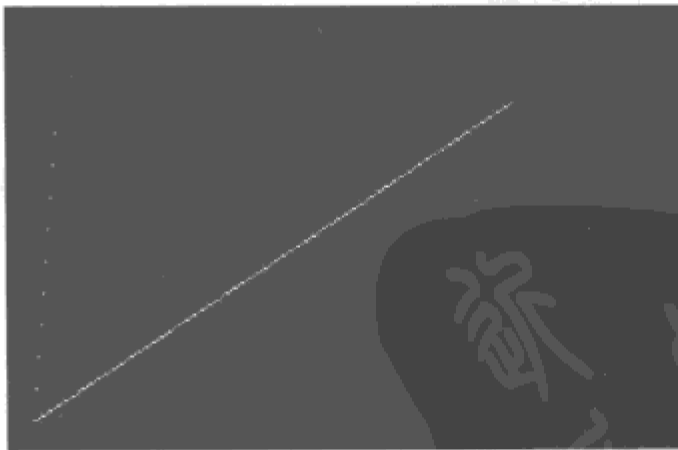


图 12-18　屏幕截图（画斜线）

12.2.13　Bresenham 算法

1962 年，正在 IBM 的圣何塞开发实验室工作的 Jack E. Bresenham 发明了一种专门使用整数运算的画线方法，在今天被视为是所有计算机画图程序的基础。它基于 3 个优化“技巧”：

(1) 将画图的方向简化成一种情况（从左到右）；

(2) 将线段的斜率简化成水平距离最大；

(3) 给方程的两边都乘以水平距离（deltax）以得到整数值。

得到的画线代码很简练也很有效。下面是根据视频模块改动得到的代码：

```
#define abs( a)      (((a)> 0) ? (a) : -(a))

void line( int x0, int y0, int x1, int y1)
{
     int steep, t ;
     int deltax, deltay, error;
     int x, y;
     int ystep;

     steep = ( abs(y1 - y0) > abs(x1 - x0));

     if ( steep )
     { // swap x and y
         t = x0; x0 = y0; y0 = t;
         t = x1; x1 = y1; y1 = t;
     }
     if (x0 > x1)
     {  // swap ends
         t = x0; x0 = x1; x1 = t;
         t = y0; y0 = y1; y1 = t;
     }

     deltax = x1 - x0;
     deltay = abs(y1 - y0);
     error = 0;
     y = y0;

     if (y0 < y1) ystep = 1; else ystep = -1;
     for (x = x0; x < x1; x++)
     {
         if ( steep) plot(y,x); else plot(x,y);
         error += deltay;
         if ( (error<<1) >= deltax)
         {

             y += ystep;
             error -= deltax;
         } // if
     } // for
} // line
```

可以给该函数添加视频模块“graphic.c”和include文件“graphic.h”中。

为了测试Bresenham算法的有效性，可以创建一个新的小项目并且再次使用“stdlib.h”库中的伪随机数发生器。下面的示例代码首先会在屏幕上画一个边框，然后会在随机生成的坐标中使用画线子程序产生100条线段。主程序还包括对S3按键（Explorer16演示板底部最左边的按键）的测试，在清屏前按下该键，那么生成的新的随机线段就会是绿色的，如图12-19所示。

```
//
// Bresenham.c
//
// Bresenham algorithm example
//
```

```
#include <p24fj128ga010.h>
#include <stdlib.h>
#include "../graphic/graphic.h"

main()
{
    int i;

    // initializations
    initVideo();     // start the state machines
    srand( 12);

    // main loop
    while( 1)
    {
        clearScreen();
        line( 0, 0, 0, VRES-1);
        line( 0, VRES-1, HRES-1, VRES-1);
        line( HRES-1, VRES-1, HRES-1, 0);
        line( 0, 0, HRES-1, 0);

        for( i = 0; i<100; i++)
            line( rand()%HRES, rand()%VRES, rand()%HRES, rand()%VRES);

     // waiting for a button to be pressed
        while( 1)
        {
            if ( !_RD6)
                break;
        } // wait

    } // main loop

} // main
```

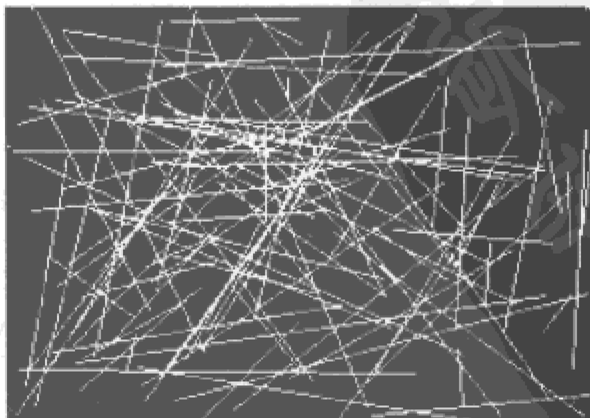


图 12-19 屏幕捕捉（Bresenham 画线算法测试）

读者会对画线算法的速度感到惊讶——即使将画线的数量提高到 1 000 条，PIC24 也能在瞬

间完成。

12.2.14 画数学函数图

有了完整的图形模块，就可以开始探索一些完全利用可视化优点的有趣程序。一个典型的应用就是描绘传感器的接收数据，或者是为了演示需要描绘计算出的简单数学函数值。

举个例子，假设一个波动的正弦函数

$y(x)=x*\sin(x)$

并假设函数 x 的画图范围在 0 到 8π之间。

为了让函数在屏幕上更好地显示，可以将输入范围重新定义为 0 到 200，输出范围定义在+75 到−75。

下面的程序例子将画出函数的 x 和 y 轴：

```
/*
** Plotting a 1D function graph
**
*/

#include <p24fj128ga010.h>
#include <math.h>

#include "../graphic/graphic.h"

#define X0 10
#define Y0 (VRES/2)
#define PI 3.141592654f

main( void)
{
    int x, y;
    float xf, yf;

    // initializations
    clearScreen();
    initVideo();

    // draw the x and y axes crossing in (X0,Y0)
    line( X0, 10, X0, VRES-10);      // y axes
    line( X0-5, Y0, HRES-10, Y0);    // x axes

    // plot the graph of the function for
    for( x=0; x<200; x++)
    {
        xf = (8 * PI / 200) * (float) x;
        yf =  75.0 / ( 8 * PI) * xf * sin( xf);
        plot( x+X0, yf+Y0);
    }

    // main loop
    while( 1);

} // main
```

如果图上的点太稀疏，可以选用画图算法将每个点与前一个点相连接，如图 12-20 所示。

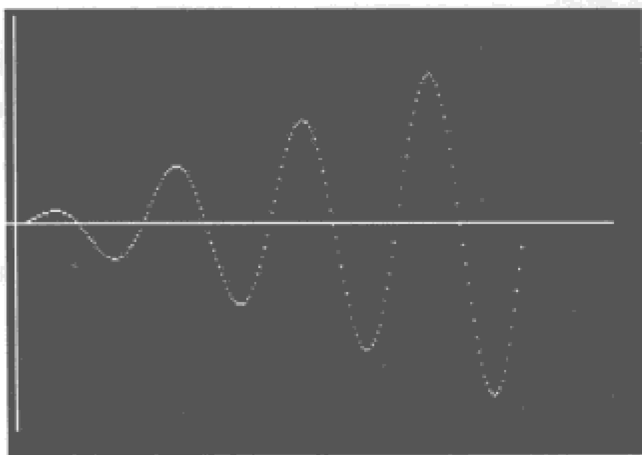


图 12-20　屏幕截图（正弦函数曲线图）

12.2.15　二维函数可视化

二维函数的绘制更加有趣或者更具有娱乐性。它增加了透视变形和计算点连接两个功能，以获得更美观的网格。

将第三个数轴压到二维图像中的最简单的办法，就是使用常说的等距投影，这种方法以最少的计算资源，提供较小的视觉失真，如图 12-21 所示。下面的方程使用三维空间点的 x、y、z 坐标，生成二维空间的 px 和 py 投影坐标。

$px = x + y/2$

$py = z + y/2$

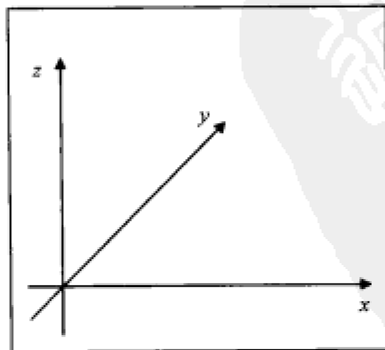


图 12-21　等距投影

为了画出已知函数 $z = f(x, y)$的三维图，要使用两个嵌套的 for 循环生成一个 x 和 y 平面等距的网格。对于计算出函数得到 z 坐标的每个点，可以使用等距投影来得到（px, py）坐标对。

然后，将新计算所得的点用线段与同一行的前一个点（前一列）相连接。另外还需要连接同列前一行的计算点，如图 12-22 所示。

尽管保持计算点在同一行上是很细微的动作，然而记录每一行的计算点坐标需要相当多的存储器空间。例如，一个 20×20 的网格，就需要存储相应的 400 个点。每个点需要 2 个整数，那么一共就需要占用 800 字（或 1 600 字节）的 RAM。实际上，对于上面的图，只需要网格“边缘”点的坐标。因此，对存储的需要可以降低到 20 对坐标，使用一个小的滚动缓冲器就可满足要求。

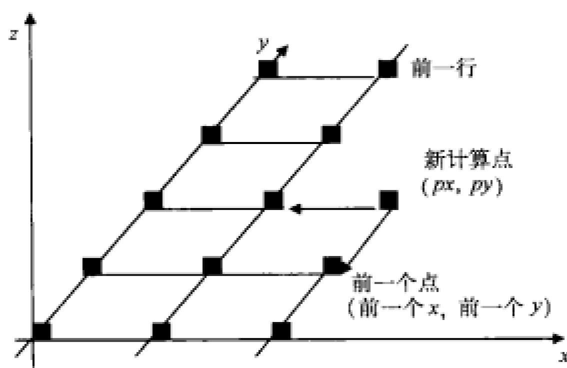


图 12-22 画出网格图以增强二维图的可视性

下面的示例代码将用于函数曲线的绘制：

$z(x, y) = 1/\mathrm{sqrt}(x^2+y^2)*\cos(\mathrm{sqrt}(x^2+y^2))$

x 和 y 的取值都在 -3π 到 $+3\pi$ 范围内。

```
/*
** Plotting a 2D function graph
**
**
*/

#include <p24fj128ga010.h>
#include <math.h>

#include "../graphic/graphic.h"

#define X0      10
#define Y0      10
#define PI      3.141592654f
#define NODES   20
#define SIDE    10
typedef struct {
        int x;
        int y;
    } point;

point edge[NODES], prev;
```

```
main( void)
{
    int i, j, x, y, z;
    float xf, yf, zf, sf;
    int px, py;

    // initializations
    clearScreen();
    initVideo();

    // draw the x, y and z axes crossing in (X0,Y0)
    line( X0, 10, X0, VRES-50);          // z axis
    line( X0-5, Y0, HRES-10, Y0);        // x axis
    line( X0-2, Y0-2, X0+120, Y0+120);   // y axis

    // init the array of previous egde points
    for( j = 0; j<NODES; j++)
    {
        edge[j].x = X0+ j*SIDE/2;
        edge[j].y = Y0+ j*SIDE/2;
    }

    // plot the graph of the function for
    for( i=0; i<NODES; i++)
    {
        // transform the x coordinate range to 0..200 offset 100
        x = i * SIDE;
        xf = (6 * PI / 200) * (float)(x-100);
        prev.y = Y0;
        prev.x = X0 + x;

        for ( j=0; j<NODES; j++)
        {
            // transform the y coordinate range to 0..200 offset 100
            y = j * SIDE;
            yf = (6 * PI / 200) * (float)(y-100);

            // compute the function
            sf = sqrt( xf * xf + yf * yf);
            zf = 1/(1+ sf) * cos( sf );
            // scale the output
            z = zf * 75;

            // apply isometric perspective and offset
            px = X0 + x + y/2;
            py = Y0 + z + y/2;

            // plot the point
            plot( px, py);

            // draw connecting lines to visualize the grid
            line( px, py, prev.x, prev.y);  // connect to prev point on same x
            line( px, py, edge[j].x, edge[j].y);

            // update the previous points
            prev.x = px;
```

```
            prev.y = py;
            edge[j].x = px;
            edge[j].y = py;
        } // for j
    } // for i

    // main loop
    while( 1);

} // main
```

构建项目，并连接显示器，PIC24 很快就输出图形，尽管这里使用有效的浮点数学运算，但视频存储器处理了 400 个点以及 800 条线段的存储，如图 12-23 所示。

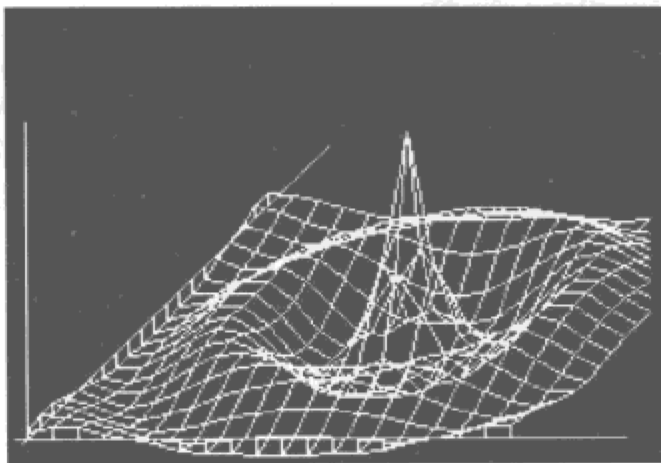


图 12-23 屏幕截图（二维函数图）

12.2.16 分形几何

“分形几何”的概念最先由一位 IBM 西北实验室数学家贝努瓦 · 曼德布罗特（Benoit Mandelbrot）在 1975 年提出的，用于定义一个有特殊性质的数学对象的巨型集合：如果形状是由无限层递归而成的，那么无论放大多少倍，得到的新图和原图都是相似的。自然界中有很多分形的例子，尽管它们的自相似性都是分布在有限范围内的。这些例子包括云朵、雪花、山脉、河道网络，甚至是人体的血管。

由于Mandelbrot集被应用于计算机可视化，因此最热门的数学分形对象可能就是它了。它被定义成一个二次函数z^2+c迭代而成的复平面的子集。特别地，迭代不会发散的复平面上的点（c）也被认作子集的一部分。尽管可以容易地证明，当z的指数大于 2 时，迭代就会发散（因此给定的点就不属于集合），只有消除这种情况才能继续迭代。问题是，当z的指数一直小于 2 时，就不能确定迭代什么时候停止，并确定集合的点。因此，通常描述Mandelbrot的计算机算法都使用近似法，只需将点假设为集合的一部分，设置一个任意的最大迭代数。

以下就是 C 语言的内部迭代程序：

```
// initialization
  x = x0;
  y = y0;
  k = 0;

 // core iteration
  do {
    x2 = x*x;
    y2 = y*y;
     y = 2*x*y + y0;
     x = x2 - y2 + x0;
     k++;
  } while ( (x2 + y2 < 4)  && ( k < MAXIT));

// check if the point belongs to the Mandelbrot set
if ( k == MAXIT) plot( x0, y0);
```

其中 x0 和 y0 是点 c 在复平面上的坐标。

将复平面的平方子集的每一个点都使用上面的迭代，就可以得到整个 Mandelbrot 集的图。理论上整个集合是包含在以原点为中心，以 2 为半径的圆盘中，因此第一个程序将会扫描复平面的一个 192×192 的网格（使用视频模块定义的屏幕最大分辨率）来覆盖这个圆盘：

```
/*
**
** Mandelbrot Set graphic demo
**
*/
#include <p24fj128ga010.h>
#include "../graphic/graphic.h"

#define SIZE  VRES
#define MAXIT 64

void mandelbrot( float xx0, float yy0, float w)
{
    float x, y, d, x0, y0, x2, y2;
    int i, j, k;

    // calculate increments
    d = w/SIZE;

    // repeat on each screen pixel
    y0 = yy0;
    for (i=0; i<SIZE; i++)
    {
        x0 = xx0;
        for (j=0; j<SIZE; j++)
        {
            // initialization
              x = x0;
              y = y0;
              k = 0;
```

```
            // core iteration
             do {
               x2 = x*x;
               y2 = y*y;
               y = 2*x*y + y0;
               x = x2 - y2 + x0;
               k++;
             } while ( (x2 + y2 < 4)  && ( k < MAXIT));

            // check if the point belongs to the Mandelbrot set
            if ( k == MAXIT) plot( j, i);

            // compute next point x0
            x0 += d;
         } // for j
         // compute next y0
         y0 += d;
      } // for i
   } // mandelbrot
   main()
   {
      float x, y, w;

      // initializations
      initVideo();          // start the state machines

      // intial coordinates lower left corner of the grid
      x = -2.0;
      y = -2.0;
      // initial grid side
      w =  4.0;

      while( 1)
      {

          clearScreen();   // clear the screen
          mandelbrot( x, y, w);

          while (1);

      } // main loop

   } // main
```

将最大迭代次数设为 64，PIC24 就会产生如图 12-24 所示的 Mandelbrot 心形，大概耗时 30 秒。

作者从儿时买了第一台个人电脑（实际上是当时是叫做“家用电脑”——是一台 Sinclair ZX Spectrum）以后，就开始玩分形程序。所以清楚地记得当时是怎样连着几个小时地守候在电脑屏幕前，等待古老而可信赖的 Z80 处理器（以超长的 3.5 MHz 速度运行）完成相同的图像。几年以后，作者买的第一台 IBM PC XT 机（8088 处理器的运行时钟只有 4 MHz）并没有好多少，尽管屏幕的分辨率在使用单色 Hercules 图形卡以后变得更好，不过晚上开始运行的程序还是要等到第二天早上才能看到结果，有时候处理时间还超过 8 小时。显然，分形图像所需的计算时

间根据所选区域以及允许的最大迭代次数不同，而有很大的区别，但是作者在首次运行这个程序的时候，还是为 PIC24 快速画出的心形出现在眼前而感到惊讶。

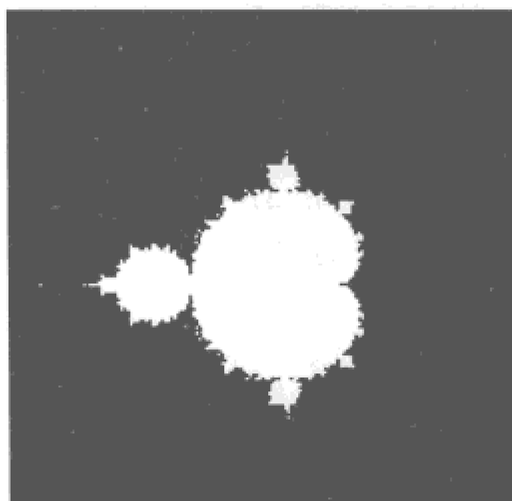


图 12-24　屏幕捕捉（Mandelbrot 集）

不过真正的乐趣才刚刚开始。Mandelbrot 集最有趣的部分在于边缘，通过放大，可以发现复平面的很多细节。要观察的不仅是属于集合的点，还有在集合边缘的那些发散的点，而它们的颜色由实际发散的速度决定，由此可以更进一步地改进图像。由于使用的是单色显示，因此只需要在每个点达到最大模数或者最大迭代次数前，每次迭代都要进行简单的黑白色转换。因为非常地简单，只需要在前一个例子中改动一行代码就可以了：

```
...
            // check if the point belongs to the Mandelbrot set
            if ( k & 1) plot( j, i);
...
```

同时，由于 Mandelbrot 集图形最好玩的就是选择新的区域并放大细节，因此可以在主程序键入简单的用户界面，利用 Explorer16 板上的四个按键来实现。可以想象图像分成四个象限，如图 12-25 所示。每个按键对应一个象限，按下，就可以放大，分辨率加倍，网格尺寸减半。

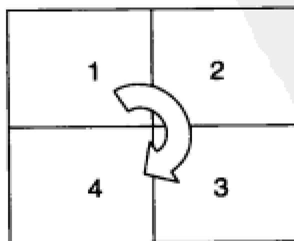


图 12-25　将屏幕分成四个象限

```
main()
{
    float x, y, w;

    // initializations
    initVideo();     // start the state machines

    // intial coordinates lower left corner of the grid
    x = -2.0;
    y = -2.0;
    // initial grid size
    w =  4.0;

    while( 1)
    {
        clearScreen();                          // clear the screen
        mandelbrot( x, y, w);                   // draw new image

        // wait for a button to be pressed
        while (1)
        {  // wait for a key pressed
            if ( !_RD6)
            {  // first quadrant
                w/= 2;
                y += w;
                break;
            }
            if ( !_RD7)
            { // second quadrant
                w/= 2;
                y += w;
                x += w;
                break;
            }
            if ( !_RA7)
            {  // third quadrant
                w/= 2;
                x += w;
                break;
            }
            if ( !_RD13)
            { // fourth quadrant
                w/= 2;
                break;
            }
        } // wait for a key
    } // main loop
} // main
```

只要读者有耐性，就会发现如图 12-26 所示的有趣图案。


图 12-26a
(+0.25 + j 0.5), w = 0.25

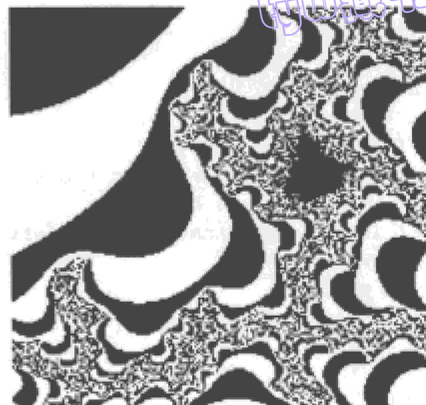

图 12-26b
(+0.37500 − j 0.57813), w = 0.01563


图 12-26c
(−1.28125 + j 0.3125), w = 0.3125

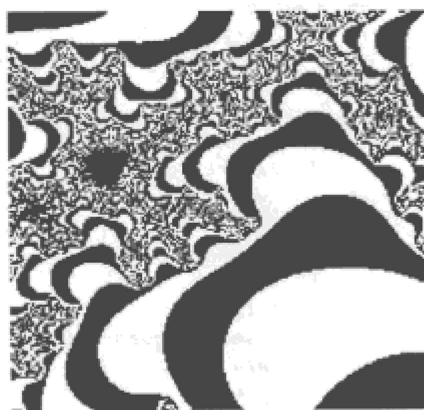

图 12-26d
(+0.34375 + j 0.56250), w = 0.03125


图 12-26e
(+0.34375 + j 0.56250), w = 0.03125

图 12-26 有趣的图案

12.2.17 文本

至此，已经掌握了很多图形的显示问题，不过读者可能还急切地想知道关于信息如何在屏幕上以文本形式显示的问题。在视频存储器中写入文本同描点和画线没什么两样，实际上有许多方法可以利用，包括使用已经生成的描点和画线函数。但是想要用最少的代码获得更高的性能，最简单的方法就是让文本的图形显示使用 8×8 的字形数组，如图 12-27 所示。每个字符可以画在 8×8 的像素框里，一个字节编码一行，8 个字节就可以编码整个字符。然后可以将 96 个基本的字母、数字和标点字符，按照它们在 ASCII 字符集中的顺序位置组合成一个数组，保存为一个 include 文件。

0	0	0	1	1	1	0	0
0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0
0	0	1	1	1	1	1	0
0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0
0	0	0	0	0	0	0	0

图 12-27 字母 A 在 8×8 字形中的表示

为了节省空间，不需要生成定义在 ASCII 集中的前 32 个代码，因为它们是过去打字机和调制器所最常使用的命令和传统的特殊同步代码。

```
//
// 8 x 8 Font definition
//

#define F_OFFS  0x20    // initial offset
#define F_SIZE  0x60    // only the first 64 characters defined so far

const char Font8x8[] = {
// 20 - SPACE
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
    0b00000000,
// 1 - !
    0b00011000,
    0b00011000,
    0b00011000,
```

```
        0b00011000,
        0b00011000,
        0b00000000,
        0b00011000,
        0b00000000,
    ...
```

注意 Font8x8[]数组由 const 定义，因为在程序的执行过程中，它的内容是保持不变的，而且最好是位于程序存储器（PIC24 的 Flash 存储器）以节省珍贵的 RAM 存储空间。

完整的“font.h”文件列表有几页纸那么多，因此这里就不详细列出了，不过读者可以在附带的 CD-ROM 中找到它。

当然，根据个人爱好，读者可以将 Font8x8[]数组改成喜欢的形式。

在屏幕上输出一个字符，就是每次一个字节地从字形数组里复制到屏幕上指定的位置。最简单的情形是，将字符与图形模块定义的 VMap（视频存储器）数组表示的字对齐。这样每行最多有 32 个字符（256/8），最多显示 24 行（192/8）。更高级的方法在任意给定的像素坐标确定字符的位置时需要完全的自由度。这种方法需要用到常说的 BitBLT（Bit Block Transfer），该操作常见于计算机图形，特别是视频游戏的设计中。下面还是使用较简单的方法，用最少的资源完成任务。

首先创建一个叫作“TextOnGPage”的项目和新的源文件“TextOnGPage.c”，包含所有在图形视频页上显示文档所需要的函数。然后定义两个整型变量用来指示光标位置：

```
int cx, cy;
```

现在就可以编写一个简单的函数，让屏幕每次在光标位置显示一个 ASCII 字符：

```
void putcV( int a)
{
    int i, *p;
    const char *pf;

    // 1. check if char in range
    a -= F_OFFS;
    if ( a < 0)          a = 0;
    if ( a >= F_SIZE)    a = F_SIZE-1;

    // 2. check page boundaries
    if ( cx >= HRES/8)          // wrap around x
    {
        cx = 0;
        cy++;
    }
    if ( cy >= VRES/8)          // wrap around y
        cy = 0;

    // 3. set pointer to word in the video map
    p = &VMap[ cy * 8 * HRES/16 + cx/2];
    // set pointer to first row of the character in the font array
    pf = &Font8x8[ a << 3];

    // 4. copy one by one each line of the character on the screen
    for ( i=0; i<8; i++)
```

```
    {
        if ( cx & 1)
        {
            *p &= 0xff00;
            *p |= *pf++;
        }
        else
        {
            *p &= 0xff;
            *p |= (*pf++)<<8;
        }
        // point to next row
        p += HRES/16;
    } // for

    // increment cursor position
    cx++;
} // putcV
```

在函数最开始的几行（程序段 1），可以验证传递给函数的字符是 ASCII 字符集中定义在字形子集的元素。如果不是，则将它变成定义范围内的第一个或者最后一个字符。另一个读者可用的策略，就是忽略该字符并退出子程序。

函数的第二部分（程序段 2）是为了让光标到达屏幕边缘时返回到下一行的开始位置，就像打字机那样。类似地，在光标到达屏幕右下角的时候，它会返回屏幕的最前面。这里还可以使用滚动特性，就是将整页的内容向上移一行，为新的文本行留出位置。

在第三部分（程序段 3），屏幕存储映射的指针计算是基于光标坐标的，而字形数组的指针计算是基于 ASCII 字符代码的。在最后的（程序段 4），循环负责一行一行地将字形图复制到视频数组。由于视频数组（VMap）是以字形式组织的（并且先显示 MSB），因此注意要以 16 位字的形式正确地传输每个字节。如果光标位置是偶数，那么所选字的 MSB 就需要用字形数据来代替。如果光标位置是奇数，那么所选字的 LSB 就需要用字形数据来代替。在循环的每一步中，视频映射（p）的指针都以 16 字（HRES/6）的幅度增加到下一行的相同位置，而字形数组（pf）的指针每次只增加 1，从而只能指向字符图的下一个字节。

为方便起见，现在创建一个函数，可以在屏幕上输出整个 NULL 终止 ASCII 字符串：

```
void putsV( unsigned char *s)
{
    while (*s)
        putcV( *s++);
} // putsV
```

同时，要记住将必要的 include 文件都编译到这个模块中：

```
#include <p24fj128ga010.h>
#include "../font/font.h"
#include "../graphic/graphic.h"
```

最后，生成一个新的 include 文件来输出新定义的函数和添加一对有用的宏：

```
/*
** Text on Graphic Page
```

```
*/

extern int cx, cy;

void putcV( int a);

void putsV( unsigned char *s);

#define Home()         { cx=0; cy=0;}
#define Clrscr()       { clearScreen(); Home();}
#define AT( x, y)      { cx = (x); cy = (y);}
```

Home() 将光标放在屏幕的左上角。

Clrscr() 通过调用图形模块中定义的函数来清屏。

AT() 将光标放在下一条 putcV 和/或 putsV 命令要求的地方。

现在要注意，与图形坐标系统不同的是，文本光标坐标系统将原点定义在屏幕左上角的开始位置，垂直坐标增加意味着向下翻页移动行。

12.2.18 测试 TextOnGPage 模块

为了能快速有效地测试新的文本模块，现在生成一个小的程序，在屏幕上显示一个由 8×8 字形字符组成的小标语：

```
/*
** Text Page Test
**
*/

#include <p24fj128ga010.h>
#include "../graphic/graphic.h"
#include "../textg/TextOnGPage.h"


main( void)
{
    int i;

    // initializations
    initVideo();     // start the state machines

    Clrscr();

    AT( 0, 0);
    putsV( "FLYING THE PIC24!");

    AT( 0, 2);
    for( i=32; i<128; i++)
    putcV( i);

    while (1);

} // main
```

将上面的文件保存为“TextOnGTest.c”并添加到项目中。要保证其他所有需要的模块也加到项目中，这包括：“graphic.c”、“graphic.h”、“font.h”、“textongpage.c”和“textongpage.h”。最后，构建项目并运行，如图 12-28 所示。



图 12-28 屏幕截图（图形页上的文本）

12.2.19 开发文本页视频

使用新建立的“TextOnGPage.c”模块，就可以在视频屏幕上显示文本和图形了。系统一共需要 6 080 字节的 RAM 用于视频映射，这对于 PIC24FJ128GA010 的 RAM 来说，会占用相当多的空间，而对于程序存储器来说只是很少的一部分，如图 12-29 所示。

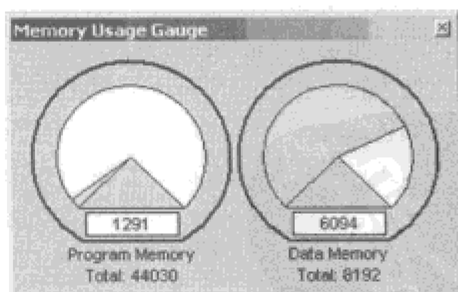


图 12-29 TextOnGTest 项目的存储器利用

如果程序的视频输出只有文本，那么这就是一个非常不高效的方案。实际上，如果使用 8×8 字形，那么每行只有 32 个字符，最多只有 24 行，即一共有 768 个字符。换而言之，如果程序使用视频作为纯文本显示，那么就浪费了 RAM 里珍贵的 5 244 个字节。在早期的计算机（包括第一台 IBM PC）时代，这是一个非常严重的问题，需要增添用户自定义硬件方案。所有早期的个人电脑系统实际上都有一个“文本页”，即一个只显示文本的视频模式，它的优点在于可以大量降低对 RAM 的需求（只有图形页的一小部分），同时可以明显提高屏幕操控性能。在文本页里，ASCII 码直接存放在视频存储区，并且通过一个连接视频扫描和定时逻辑的硬件设备（叫做字形发生器）可以很快地转换成图形字形。这样，一页 768 个字符（跟前一个项目相同）

所需的存储空间就只需 768 字节，这个值只有图形显示方案所需存储空间的 10%左右。

这听起来像是一个新的挑战。在下一个项目里，将开发具有更高 RAM 使用效率的视频方案，并应用于纯文本显示。这就需要再次回到核心图形视频模块的状态机的初始化定义上。实际上，可以保留它大部分的结构，只对一些重要的地方进行改动。所有组成水平和垂直同步信号的元素保持不变。同样地，水平行的结构保持不变，直到向 SPI1 模块发送数据以串行化之前。在图形显示中，首先需要将存储器映射的每个字读取出来，然后压进 SPI 缓冲器，在文本页视频程序中，需要每次操作一个字节并插入一个转换环节。将 Font8×8[]数组作为查找表，用于将文本页上的 ASCII 码（现在 VMap 定义为字节数组）转换成要发送到 SPI 缓冲的串行化图像。通常，这个转换可以用下面的表达式来表示：

```
lookup = Font8x8[ *VPtr * 8  + RCount];
```

其中 VPtr 是当前字符在文本页数组中的指针，而 RCount 是 0 到 7 的计数值，用于追踪由一行文本组成的视频行（每个文本行由 8 个视频行组成）。

在实践中，事情会稍微复杂一些。由于 SPI 模块必须每次输入 16 位数据，因此在执行两个连续的查找后，必须将两个字符组合成一个字：

```
lookup1 = Font8x8[ *VPtr++ * 8  + RCount];
lookup2 = Font8x8[ *VPtr++ * 8  + RCount];
SPI1BUF = ( 256 * lookup1 + lookup2);
```

将上面的操作重复 8 次，可以把整个 SPI 缓冲器填满。

现在，在 OC4 中断服务子程序的短短几毫秒的时间内还有很多工作要做。即使编译已经得到最大程度的优化（而在本书中从没有用过任何优化），但是要在规定时间（小于 25 μs）内完成还是很困难的。简单说来就是操作太多，而且在查表过程中有很多附加的操作。幸好，有些事情是可以变通的。实际上，可以重组 Font 数组的排列方式。尽管将数组按每个字符 8 行的方式排列和采取连续处理会比较方便，不过为了简化查找表达式，最好还是将它按其他的方式排列。换而言之，数组可以是先放入每个字符字形的第一个字节，然后是每个字形的第二个字节，以此类推。那么上面的表达式可以应用新的重组字形 RFont 改写为如下形式：

```
lookup1 = RFont[  (RCount * F_SIZE) + *VPtr++ ];
lookup2 = RFont[  (RCount * F_SIZE) + *VPtr++ ];
SPI1BUF = ( 256 * lookup1 + lookup2);
```

它最大的改进是 RCount*F_SIZE 是一个常量偏移量，可以通过下面的表达式给出的偏移量来获得字符内的指针：

```
FPtr = &RFont[ RCount * F_SIZE];
```

这个值是可以预先计算的（在 Timer3 中断服务子程序里），可以明显地节省很多的计算量。新的查找表达式可以简化为：

```
lookup1 = FPtr[ *VPtr++ ];
lookup2 = FPtr[ *VPtr++ ];
SPI1BUF = ( lookup1 << 8 + lookup2);
```

现在有了在那几毫秒时间完成查找的可能，不过还不能满足于此。子程序中每纳秒一次的

计数和 OC4 中断服务子程序的调用一样重要和频繁。实际上最终的优化技巧就是对几个重要的步骤进行人工汇编。如果假设字形指针（FPtr）已经放入 W2 工作寄存器，而视频存储指针（VPtr）已经放入 W1 工作寄存器，那么整个查找序列就可以使用 3 条汇编指令完成：

```
mov.b [w1++], w0          // w0 = *VPtr++       (8 bit)
ze  w0, w0                // extend w0 to a 16 bit integer
mov.b [w2+w0], w3         // w3 = FPtr[ w0] = FPtr[ *VPtr++] = lookup1
```

对 lookup2 重复相同的指令是很简单的，将两个值合成到一个字只需要一个移位操作：

```
sl w3, #8, w3             // shift W3 8 bits to the left (*256)
```

然后相加：

```
add w0, w3, w0            // add (lookup1*256) and lookup2
```

然后将所有的代码放到一个叫作 DECODE()的宏里面：

```
#define DECODE( sfr) \
    asm volatile ( "mov.b [w1++], w0" );     \
    asm volatile ( "ze w0, w0");             \
    asm volatile ( "mov.b [w2+w0], w3");     \
    asm volatile ( "sl w3,#8,w3" );          \
    asm volatile ( "mov.b [w1++], w0" );     \
    asm volatile ( "ze w0, w0");             \
    asm volatile ( "mov.b [w2+w0], w0");     \
    asm volatile ( "ze w0, w0");             \
    asm volatile ( "add w0, w3, w0");        \
    asm volatile ( "mov w0, %0" : "=U"((sfr)));
```

这里使用类 volatile 来确保编译器将来引入优化机制后，不会改变内联汇编代码的顺序和位置。同时最后一行代码看起来有些奇怪。实际上，这是 C30 编译器提供的一个内联语法特性，可以将 C 变量名作为参数传递给 asm() 函数。特殊的符号："=U"()说明括号中的操作符作为输出数据传递。

现在修改 OC4 中断子程序以充分利用改进的字形表查找：

```
void _ISRFAST _OC4Interrupt( void)
{
    // prepare pointers
    volatile asm ( "mov %0, w2" ::"U" (FPtr) );    // w2 = FPtr
    volatile asm ( "mov %0, w1" ::"U" (VPtr));     // w1 = VPtr

    // inline text to font translation * 8 words
    DECODE( SPI1BUF);
    DECODE( SPI1BUF);
    DECODE( SPI1BUF);
    DECODE( SPI1BUF);
    DECODE( SPI1BUF);
    DECODE( SPI1BUF);
    DECODE( SPI1BUF);
    DECODE( SPI1BUF);
    __asm__( "mov w1, %0" :"=U" (VPtr));      // update VPtr
```

```
    if ( --HCount > 0)
    {   // activate again in time for the next SPI load
            OC4R += ( PIX_T * 8 * 16);
            OC4CON = 0x0009;      // single event
    }

    // clear the interrupt flag
    _OC4IF = 0;

} // OC4Interrupt
```

正如前面所说的，Timer3 中断服务子程序的修改是很细微的，只需要准备一对用于文本行的排序的指针，以及预先计算好的字形偏移量：

```
void _ISRFAST _T3Interrupt( void)
{
    // Start a Sync pulse
    SYNC = 0;

    // decrement the vertical counter
    VCount--;

    // vertical state machine
    switch ( VState) {
        case SV_PREEQ:
            // horizontal sync pulse
            OC3R = HSYNC_T;
            OC3CON = 0x0009;      // single event
            break;

    case SV_SYNC:
        // vertical sync pulse
        OC3R = H_NTSC - HSYNC_T;
        OC3CON = 0x0009;      // single event
        break;

    case SV_POSTEQ:
        // horizontal sync pulse
        OC3R = HSYNC_T;
        OC3CON = 0x0009;      // single event
        // on the last posteq prepare for the new frame
        if ( VCount == 0)
        {
            LPtr = VMap;
            RCount = 0;
        }
        break;

    default:
    case SV_LINE:
        // horizontal sync pulse
        OC3R = HSYNC_T;
        OC3CON = 0x0009;      // single event

        // activate OC4 for the SPI loading
```

```
            OC4R = HSYNC_T + BPORCH_T;
            OC4CON = 0x0009;    // single event
            HCount = 3;         // reload counter

            // prepare the font pointer
            FPtr = &RFont[ RCount * F_SIZE];
            // prepare the line pointer
            VPtr = LPtr;

            // Advance the RCount
            if ( ++RCount == 8)
            {
                RCount = 0;
                LPtr += COLS;
            }
    } //switch

    // advance the state machine
    if ( VCount == 0)
    {
        VCount = VC[ VState];
            VState = VS[ VState];
     }

    // clear the interrupt flag
    _T3IF = 0;

} // T3Interrupt
```

视频初始化子程序还需要多一个步骤，因为字形数组需要重组，如前面所说的：

```
// prepare a reversed font table
for (i=0; i<8; i++)
{
    p = Font8x8 + i;
    for (j=0; j<F_SIZE; j++)
    {
        *r++ = *p;
        p+=8;
    } // for j
}  // for i
```

为方便起见，这个字形表作为 RAM 里的另一个数组，里面的数据按新的顺序排列，最后就是重组“font.h”文件定义，所以 Font8×8 数组已经是用新的优化顺序定义的，没有浪费 RAM 空间，也不需要在视频初始化期间执行转换工作。

回到图形界面发现，256×192 像素屏幕是经过对屏幕分辨率和存储器使用率的综合考虑而得到的，因为它可以给应用程序留出 2 KB 的 RAM 空间。现在这个平衡大大地改变了。对于 24 行 32 列的显示，视频模块只使用了 786 字节，因此实际上可以稍微扩展一下分辨率。水平分辨率是最需要提升的。大多数的视频终端都使用 25 × 80 格式，而显示的文本平均每行不少于 60 个字符。当 RAM（25 行 × 80 列=2 000 字符）有足够的空间时，就需要用到 NTSC 视频规定的极限了。正如本章开始时候介绍的，NTSC 视频合成信号的最大信号带宽是 4.2 MHz，而产

生可视行图像的波形部分只有 52 μs 宽。这就决定了最大水平分辨率理论值是 436 个像素，对于 8×8 字形来说，最多提供 54 列。实际上还是选择小一点的值比较好，此外要充分利用至今运用得很好的 SPI FIFO 机制，应该选择 16 的倍数值。在图形模块中，可以使用 2 个 128 像素的连续块来填充 SPI FIFO 缓冲器，而在文本页模块，现在可以加入第三个模块，将总的水平分辨率提高到 48 字节。要注意这样需要将 SPI 时钟的预分频器值转换为更高的频率模式 (PIX_T=2)。

对于垂直分辨率的选择就很灵活了，由于 NTSC 标准规定的 262 行有 253 行可以用于实际图像。可以简单地定义成 25 行文本（最多 200 行)。

文本页模块产生 25 行 48 列的显示，一共只有 1 200 字节。与图形页方法相比，由于大大减低了 RAM 的使用，所以可靠性会有相当大的提高。

下面的代码用以完成新“文本”视频模块的常量和定义：

```
/*
** TextPage.c
**
** Text Page video module
**
*/

#include <p24fj128ga010.h>
#include "../Text/TextPage.h"
#include "../font/font.h"

// I/O definitions
#define SYNC      _LATG0   // output
#define SDO       _RF8     // SPI1 SDO

// calculates the NTSC video parameters for the vertical state machine
#define V_NTSC     262     // total number of lines composing a frame
#define VRES      (ROWS*8) // desired vertical resolution (<242)
#define VSYNC_N    3       // V sync lines
// count the number of remaining black lines top+bottom
#define VBLANK_N     (V_NTSC -VRES - VSYNC_N)

#define PREEQ_N    VBLANK_N /2           // pre equalization + bottom blank
#define POSTEQ_N   VBLANK_N - PREEQ_N    // post equalization + top blank lines

// definition of the vertical sync state machine
#define SV_PREEQ      0
#define SV_SYNC       1
#define SV_POSTEQ     2
#define SV_LINE       3

// calculates the NTSC video parameters for the horizontal state machine
#define H_NTSC   1018     // total number of Tcy in a line (63.5us)
#define HRES    (COLS*8)  // desired horizontal resolution (divisible by 16)
#define HSYNC_T   72      // Tcy in a horizontal sync pulse (4.7us)
#define BPORCH_T 90       // Tcy in a back porch (4.7us)
#define PIX_T     2       // Tcy in each pixel
#define LINE_T    HRES * PIX_T         // Tcy in each horizontal image line
```

```
// Text Page array
unsigned char  VMap[ COLS * ROWS];
unsigned char *VPtr, *LPtr;

// reordered Font
unsigned char RFont[F_SIZE*8];
unsigned char *FPtr;

volatile int HCount, VCount, RCount, VState, HState;

// next state table
int VS[4] = { SV_SYNC, SV_POSTEQ, SV_LINE, SV_PREEQ};
// next counter table
int VC[4] = { VSYNC_N,  POSTEQ_N,    VRES,  PREEQ_N};
```

为 TextOnGPage 项目开发的相同子程序，现在可以直接添加到这个项目中。

```
void haltVideo()
{
    T3CONbits.TON = 0;   // turn off the vertical state machine
} //haltVideo


void initScreen( void)
{
    int i, j;
    char *v;

    v = VMap;

    // clear the screen
    for ( i=0; i < (ROWS); i++)
        for ( j=0; j < (COLS); j++)
            *v++ = 0;
} //initScreen


int cx, cy;

void putcV( int a)
{
    // check if char in font range
    a -= F_OFFS;
    if ( a < 0)          a = 0;
    if ( a >= F_SIZE)    a = F_SIZE-1;
    // check page boundaries
    if ( cx >= COLS)         // wrap around x
    {
        cx = 0;
        cy++;
    }
    cy %= ROWS;              // wrap around y

    // find first row in the video map
    VMap[ cy * COLS + cx] = a;
```

```
    // increment cursor position
    cx++;
} // putcV


void putsV( unsigned char *s)
{
    while (*s)
        putcV( *s++);
} // putsV


void pcr( void)
{
    cx = 0;
    cy++;
    cy %= ROWS;
} // pcr
```

将新的项目文件保存为“TextPage.c”，同时生成新的 include 文件“TextPage.h”。

```
/*
** TextPage.h
**
** Text Page Video Module
**
*/


#define ROWS     25      // rows of text
#define COLS     48      // columns of text


// Text Page array
extern unsigned char  VMap[ COLS * ROWS];

// initializes the video output
void initVideo( void);

// stops the video output
void haltVideo();

// clears the video map
void initScreen( void);

// cursor
extern int cx, cy;

void putV( int a);

void putsV( unsigned char *s);

void pcr( void);
```

```
#define home()        { cx=0; cy=0;}
#define clrscr()      { initScreen(); home();}
#define AT( x, y)     { cx = (x); cy = (y);}
```

12.2.20 测试文本页性能

为了测试新文本页视频模块，可以修改在前面章节已经出现过的一个例子：黑客帝国演示项目。当时使用的是异步串行通信模块（UART1）与 VT100 计算机终端进行通信（或者更像是 PC 运行 HyperTerminal 程序，并配置成以前的 DEC 终端 VT100 协议）。现在要将用于发送字符到串行端口的 putcU 子程序修改成直接调用视频接口的 putcV 子程序。

首先创建一个项目叫作“Matrix2”，然后添加所有必要的模块，包括：rand.c、rand.h、textpage.c 及 textpage.h，最后是调用“matrix2.c”或者“the-matrix-reloaded.c”的新主程序模块。

```
/*
**  The Matrix Reloaded
**
*/

#include <p24fj128ga010.h>
#include "../random/rand.h"
#include "../Text/TextPage.h"

#define COL     40
#define ROW     24
#define DELAY 12000
#define pcr() {cx = 0; cy++;}

main()
{
    int v[40];  // vector containing lengh of each string
    int i,j,k;

    // 1. initializations
    T1CON = 0x8030; // TMR1 on, prescale 256, Tcy/2

    initVideo();
    clrscr();         // clear the screen
    randomize( 12); // start the random number sequence

    // 2. init each column lenght
    for( j =0; j<COL; j++)
        v[j] = rand()%ROW;

    // 3. main loop
    while( 1)
    {
        home();

        // 3.1 refresh the screen with random columns
        for( i=0; i<ROW; i++)
        {
```

```
            // refresh one row at a time
            for( j=0; j<COL; j++)
            {
                // print a random character down to each column lenght
                if ( i < v[j])
                    putcV( 'A' + (rand()%32));
                else
                    putcV(' ');
            } // for j
            pcr();

        } // for i

            // 3.1.1 delay to slow down the screen update
            TMR1 =0;
            while( TMR1<DELAY);
        // 3.2 randomly increase or reduce each column lenght
        for( j=0; j<COL; j++)
        {
            switch ( rand()%3)
            {
                case 0: // increase length
                        v[j]++;
                        if (v[j]>ROW)
                            v[j]=ROW;
                        break;

                case 1: // decrease length
                        v[j]--;
                        if (v[j]<1)
                            v[j]=1;
                        break;

                default:// unchanged
                        break;
             } // switch
        } // for

    } // main loop
} // main
```

在保存和构建项目后，在已经连接视频设备的 Explorer16 演示板上运行程序。读者可以发现，屏幕的更新速度快了很多，这是由于程序现在是直接访问视频存储器，而并没有串行连接限制信息的传送（最快可以达到如前一个演示项目所示的 115 200 bps）。同时，因为现在每个字符都放在了视频存储器中，可以在原来的位置读取和操作，因此一些新的技巧可以使得视频更类似于电影，具有滚动效果。

除了视觉的冲击，现在关注的是计算处理器在新的视频子程序中执行快速字形转换的实际开销。为此，MPLAB SIM 软件仿真器再次成为选用工具。和前面章节的操作一样，当执行 3 个中断服务子程序中的任意程序代码时，都使用 PORTA 的一个引脚（RA0）作为的标志信号：

```
void _ISRFAST _T3Interrupt( void)
{
  _RA0=1;
...
    _RA0=0;
} // T3Interrupt

void _ISRFAST _OC3Interrupt( void)
{
    _RA0=1;
...
    _RA0=0;
} // OC3Interrupt

void _ISRFAST _OC4Interrupt( void)
{
    _RA0=1;
...
    _RA0=0;
} // OC4Interrupt
```

记得将 TRISA 寄存器的初始化添加到 initVideo() 函数或者主程序中，以对 RA0 引脚输出使能。然后，将 RG0（用于产生同步脉冲）和 RA0 引脚添加到逻辑分析器窗口通道。

重构项目，运行一小会儿，可以看到最先的几行是最差情景的图像行，而其中大部分的工作都是由中断服务子程序完成的，如图 12-30 所示。

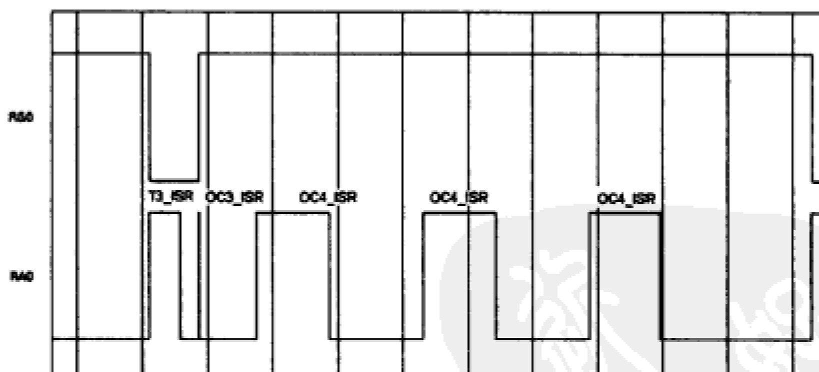


图 12-30 逻辑分析器窗口（测量文本页视频模块的消耗）

利用光标特性，现在可以测量在每个水平行期间，每个中断服务子程序所需要的时钟周期数。只有 StopWatch 工具才能够给出准确的周期计算，而逻辑分析器窗口通过少量的工作也可以提供较好的近似结果。作者的计算显示，每 1 018 个时钟周期中，视频模块的中断服务子程序就占用了 384 个，大概接近处理器有效计算能力的 38%。这几乎是图像视频模块子程序的两倍，20%的差值足以大量地减低 RAM 存储需求和提高分辨率，而这些都得益于纯文本输出。

12.3 飞后小结

本章讨论了如何使用最小的三电阻硬件接口来产生视频输出，学习了如何使用 4 个模块构

建期望的复杂机制来产生正确的 NTSC 合成视频信号。一个 16 位定时器用于生成基本的水平同步周期。两个输出比较模块提供中间的定时参考，最后 SPI 模块的增强模式用于产生 8 层的 16 位 FIFO，用于视频数据串行化。在开发了用于描点、画线的基本图形函数后，又研究了图形视频输出的可行性，包括一维和二维函数图形。在简单介绍了分形几何后，又讨论了文本显示的问题。首先是使用文本加载到图形页上的方法，然后开发了只用于文本显示的优化视频模块。

12.4　提示与技巧

最后，为了完成本书对图形世界的探索，应该在视频输出库里加入动画。为了让动作更流畅，同时避免屏幕上不断地出现恼人的闪烁，需要掌握一种叫做"双缓冲"的技术。这需要任何时候都有两个图形缓冲可同时使用。其中一个是"活动"缓冲，它的内容是要显示在屏幕上的，而另一个叫做"隐藏"缓冲，是要消退的。当第二个缓冲的内容完全消退，那么这两个缓冲就会互换。第一个缓冲不再可见，被清空，然后绘图过程重新开始。在这里，唯一限制这种技术使用的是 RAM 存储器的容量。为了使得两个图像缓冲器可以放到 PIC24FJ128GA010 的 8 KB 存储器中，并且还要为变量和栈保留一定的空间，那么就需要降低图像的分辨率。例如一对 160×160 的图像缓冲器，就只需要 3 200 字节：

```
int _FAR V1Map[VRES * (HRES/16)];
int _FAR V2Map[VRES * (HRES/16)];
```

程序只需以下改动。

- 将 VMap[] 数组的直接寻址，改为指针寻址。
- 刷新屏幕的中断驱动状态机使用活动缓冲器指针：

```
int *VA;
```

- 将描点和画线函数使用隐藏缓冲器指针：

```
int *VH;
```

两个缓冲器的互换可以由两个指针的互换实现：

```
void swapV( void)
{
    int * V;
    while ( VCount != 1);        // wait until the end of the frame
    V = VA; VA = VH; VH = V;     // at the next VSynch it will swap the screen
} //swapV
```

注意转换不能在帧的中间执行，但是可以是在帧的开始或者结束时同步进行。

12.5　练习

(1) 取代"write.c"函数，重定向"stdio.h"库函数，输出结果到文本/图形屏幕。

(2) 增加支持 PS/2 键盘输入，提供更为完善的控制平台。

12.6 推荐书目

- R.Koster, 2004
  A Theory of Fun for Game Design
  Paraglyph Press
  游戏的设计必须是严肃认真的。但也许不可能？

12.7 网上链接

- http://en.wikipedia.org/wiki/Zx_spectrum
  Sinclair ZX Spectrum 是 20 世纪 80 年代初最早出现的个人电脑之一（当时通常叫作家用电脑）。它的图像能力和本章使用的图像库非常相似。尽管它使用了几个用户自定义逻辑设备来提供视频输出，可是它的处理能力还不到 PIC24 的四分之一。尽管它的色彩能力很低（16 色，分辨率为 8×8），不过还是吸引了许多程序员来开发视频游戏。



第 13 章　大容量存储

本章内容

- ▶SD™/MMC 卡物理接口
- ▶与 Explorer16 板的接口
- ▶开始一个新项目
- ▶选择 SPI 操作模式
- ▶在 SPI 模式发送命令
- ▶完成 SD/MMC 卡初始化
- ▶从 SD/MMC 卡读取数据
- ▶向 SD/MMC 卡写入数据
- ▶使用 SD/MMC 接口模块

一架飞机的重量和性能之间的关系是大部分飞行员和普通人都知道的。如果机翼上的负重过多，那么起飞时间就会变长——变得很长以至于没有足够的跑道来支持，也就根本起飞不了。哎呀！

更常见的问题就类似于携带重物一样。搭乘飞机上的亲朋好友，就如同远足的行囊，看起来好像能塞进背包里的东西实际上未必能塞进去。作为飞行员，是不能仅凭猜想来决定这个问题的。飞行员必须设计重量和平衡的表格，在必要时，还需要称重，决定保留什么、丢弃什么，是丢弃行李还是丢弃燃料。但有一件事情是不建议去做的，那就是让你的好友去过秤。

13.1　飞行计划

很多嵌入式控制应用都需要更大的永久性数据存储空间，远超过前面章节中出现过的常用串行 EEPROM 设备的容量，更大于微控制器内部的 Flash 程序存储器空间的。所需的增量可能达几个数量级，上百兆字节，甚至可能到吉字节。如果读者有一台数码相机，或者 MP3，甚至是手机，就应该对消费类多媒体设备的容量需求以及多种存储技术略知一二。虽然硬盘占用的空间变小了并且对电源的需求也变少了，但是多种固态方案充斥着市场（例如基于 Flash 技术的 CompactFlash®、SmartMedia™、安全数码卡（SD）、Memory Stick®等）。由于被消费市场大量使用，它们的价格已经降低到可以集成在嵌入式控制设备上的程度了。

在本章中，会介绍如何以最少的处理器资源，把一个常见又便宜的大容量存储设备连接到 PIC24 微控制器上。

13.2　飞行

由于每一种大容量存储技术都是为某种特别的应用设计的，因此都有它的优点和缺点。大容量存储媒介的选用应该考虑以下标准：

- ❑ 存储器以及所需连接器的广泛适用性；
- ❑ 物理接口（串行）需要的引脚数要少；

- 大存储容量；
- 开发的技术规范；
- 易于实现；
- 存储器和所需连接器的价格便宜。

安全数码（SD）标准满足以上的所有要求，也是当今数码相机和很多多媒体消费电子最常用的大容量存储媒介。SD 卡是标志着多媒体卡（MMC）的一次技术革新，两者在电子结构和机制上还保持着部分兼容性。安全数码卡协会（SDCA）拥有和控制 SD 卡的技术标准，并且要求所有设计、改进、生产或者销售 SD 卡的公司都必须加入到组织中。目前，一个普通的会籍需要缴交 2 000 美元的年费。而另一方面，多媒体卡协会（MMCA）并不要求强制的入会，不过 MMC 的技术规范至少要 500 美元。因此，这两种技术离免费和开放还很遥远。幸好，SDCA 公开了一个 SD 技术的规范子集，称为"简化物理规范"。这个信息可以帮助读者理解最基本的 SD/MMC 存储器技术，并且开始设计 PIC24 大容量存储接口。

13.2.1 SD/MMC 卡物理接口

SD 卡只需要 9 条电气连接和一个 SD/MMC 兼容连接器（可以在很多网站上以低于 2 美元的价格买到，连接器的引脚如图 13-1 所示，另外还需要一对引脚用于插入检测和写保护开关检测。它有两种通信模式：第一种（叫做 SD 总线），源自 SD/MMC 标准，需要一个半字节（4bit）宽的总线接口；第二种是串行的，基于常见的 SPI 总线标准。正是第二种模式，让 SD/MMC 大容量设备用于所有的嵌入式控制，因为大多数的微控制器都有硬件 SPI 接口或者可以容易地使用少量的 I/O 来仿真（位操作）。最后，SD/MMC 的物理技术规范指出，2.0V 到 3.6V 的操作电源范围最适合当今高级 CMOS 处理的微控制器，也就是 PIC24 系列使用的电源范围。

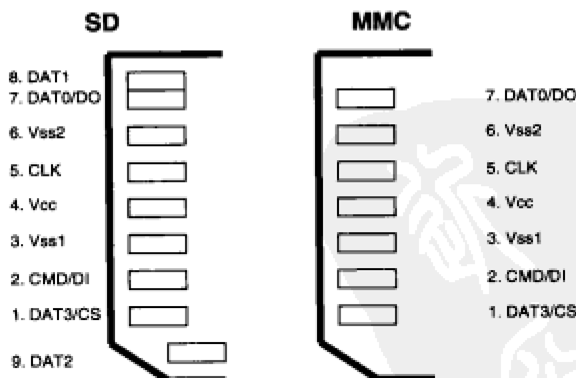


图 13-1 SD 卡和 MMC 卡连接器引脚

13.2.2 连接 Explorer16 演示板

不过，尽管 SPI 接口需要的电气连接数量很少，市场上所有的 SD/MMC 卡连接器都只是为表面贴装设计的，也就是几乎不可能用在 Explorer16 演示板的原型区。为了本章的课程顺利进行，在下面的课程中使用大容量存储设备，扩展板的完整图表和 PCB 布局信息已经放在网站 http://www.flyingthePIC24.com 上了。扩展板的其他接口也会出现在本书后面的章节中。

由于在前面的章节中，已经使用过 SPI 的第一个外部模块来生成音频输出，而该应用不允许资源共享，因此对于第二个 SPI 模块（SPI），我们会通过不同的片选信号，让 SD 卡接口和 EEPROM 接口共享。除了常用的 SCK、SDI 和 SDO 引脚外，还会给 SD/MMC 连接器中不常用的引脚（预留给 4bit 宽 SD 总线接口）提供上拉电阻，另外还有两个引脚用来给插卡检测和写保护提供信号，如图 13-2 所示。

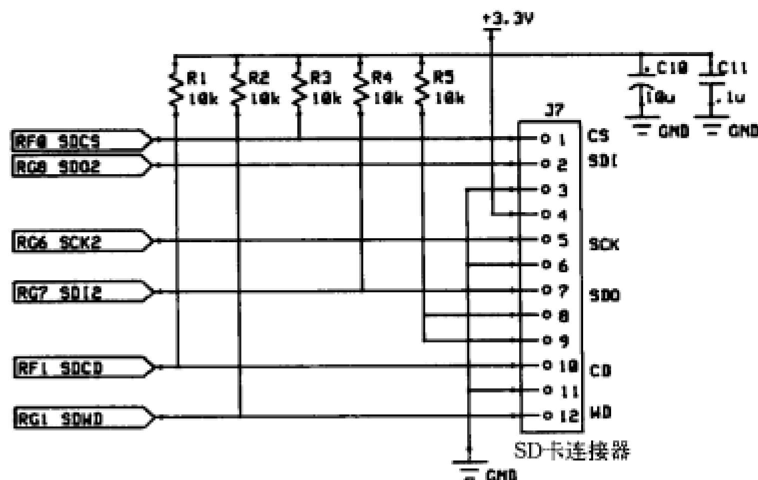


图 13-2　SD/MMC 卡与 Explorer16 演示板接口

13.2.3　开始一个新项目

（使用常用列表）生成一个新项目，开始对必要的 I/O 和 SPI2 模块编写基本的初始化子程序：

```
/*
** SD card interface
**
*/

#include <p24fj128ga010.h> // pin out definitions

#define SDWD    _RG1    // Write Protect input
#define SDCD    _RF1    // Card Detect input
#define SDCS    _RF0    // Card Select output

void initSD( void)
// initializes the I/Os and peripheral modules (SPI2)
{
    SDCS = 1;           // default Card not-selected (high)
    _TRISF0 = 0;        // make only Card select an output pin

    // init the spi module for a slow (safe) clock speed first
    SPI2CON1 = 0x013c;  // CKE=1, SMP=0, CKP=0, prescale 1:64

    SPI2STAT = 0x8000;  // enable the SPI2 peripheral
}   // initSD
```

特别是在 SPI2CON1 寄存器中，需要将 SPI 模块设置成主控制模式工作，并且需要有正确的时钟极性、时钟沿、输入采样点和初始时钟频率。闲时，时钟输出（SCK）必须是使能的并且是低电平的。SDI 输入的采样点必须位于中间。频率由两个预分频器（初级和二级）的均值除以主处理器的周期时钟（Tcy）来产生 SPI 时钟信号。在通电和 SD 卡初始完毕前，需要降低时钟速度到安全设置（小于 400 kHz），因此，需要将初级预分频器设置成 1：64，保持 250 kHz 的时钟信号。这只是一个临时的值，在发送几条命令行后，就需要大幅度提高通信速度。

> **注解** 只有控制卡选择信号的 RF0 引脚，需要人工设置为输出引脚；而 RG6 和 RG8（分别对应引脚 SCK2 和 SDO2）会在 SPI2 外设使能的时候自动地设为输出。

13.2.4 选择 SPI 操作模式

当 SD/MMC 插入到连接器并通电时，就进入了默认的通信模式——SD 总线模式。为了通知卡将要使用 SPI 模式通信，只需发送卡选信号（向 SDCS 引脚发送低电平），发送第一个 RESET 命令。可以假定，当进入 SPI 模式时，卡不能再回到 SD 总线模式，除非重新上电。也就是说，如果卡已从插槽拔出，然后再插入，那么就需要确保初始化子程序或者至少重启命令被重新执行，以回到 SPI 模式。用户可以通过检查连接 CD 线的 RF1 引脚，来检测卡的状态。

13.2.5 在 SPI 模式发送命令

在 SPI 模式，命令打包成 6 字节发送到 SD/MMC 卡，而 SD 卡的响应是以不定长的多字节数据块形式反馈。因此，和存储卡的通信就是使用基本 SPI 子程序每次发送和接收（正如前面章节看到的，这两个操作是完全一样的）1 字节信息：

```
// send one byte of data and receive one back at the same time
unsigned char writeSPI( unsigned char b)
{
    SPI2BUF = b;                    // write to buffer for TX
    while( !SPI2STATbits.SPIRBF);   // wait for transfer to complete
    return SPI2BUF;                 // read the received value
}// writeSPI
```

为了让代码更易读且易于修改，我们同时定义两个宏，将相同的 writeSPI()函数屏蔽为纯粹的 readSPI()函数或者时钟输出函数 clockSPI()。两个宏都需要发送一个哑字节数据（0xFF）：

```
#define readSPI()    writeSPI( 0xFF)
#define clockSPI()   writeSPI( 0xFF)
```

为了发送命令，需要选择卡（SDCS 低），然后通过 SPI 端口发送由三部分构成的数据包。SPI 模式 FSD/MMC 卡的命令格式见图 13-3。

<table>
<tr><th colspan="8">BYTE 1</th><th colspan="8">BYTE 2</th><th colspan="8">BYTE 3</th><th colspan="8">BYTE 4</th><th colspan="8">BYTE 5</th><th colspan="8">BYTE 6</th></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td></td><td></td><td colspan="6">命令</td><td colspan="32">地址</td><td colspan="8">CRC</td></tr>
</table>

图 13-3 SPI 模式下 SD/MMC 卡的命令格式

第一部分是一个字节，包含有命令索引。下面的定义包含本项目中要用到的所有命令：

```
// SD card commands
#define RESET             0     // a.k.a. GO_IDLE (CMD0)
#define INIT              1     // a.k.a  SEND_OP_COND (CMD1)
#define READ_SINGLE       17    // read a block of data
#define WRITE_SINGLE      24    // write a block of data
```

命令索引后面是 32 位的存储器地址。unsigned long 整数会先发送 MAS。为方便起见，需要定义一个新的类型表示地址域，叫做 LBA，借用其他大容量存储的术语来表示大量数据的地址。

```
typedef unsigned long LBA;      // logic block address, 32 bit wide
```

命令包以一个字节的 CRC 结束。循环冗余校验（CRC）通常使用 SD 总线模式来保证每个命令和每个数据块都正确发送。不过，当转换到 SPI 模式（通过发送 RESET 命令）时，CRC 保护会自动屏蔽，因为假设卡是直接可靠地连接到主机上的，这里使用的是 PIC24。由于这个默认的性质，在代码中，可以简单地用预先的计算值来代替 CRC 的计算。RESET 命令的 CRC 代码，会忽略 CRC 域的子集命令。下面是 sendSDCmd() 函数的第一部分：

```
int sendSDCmd( unsigned char c, LBA a)
// sends a 6 byte command block to the card and leaves SDCS active
{
    int i, r;

    // enable SD card
    SDCS = 0;

    // send a comand packet (6 bytes)
    writeSPI( c | 0x40);     // send command + frame bit
    writeSPI( (unsigned char) a>>24);       // msb of the address
    writeSPI( a>>16);
    writeSPI( a>>8);
    writeSPI( a);            // lsb
// NOTE only CMD0-RESET requires an actual CRC (once in SPI mode CRC is disabled)
    writeSPI( 0x95); // send CRC of RESET, for all other cmds it's a don't care
```

发送 6 字节命令到卡后，需要在循环中等待一个响应字节数据（实际上，是一直发送哑数据为 SPI 端口提供时钟）。响应数据是 0xFF（SDI 线会保持高），直到存储卡准备好发送正确的响应代码为止。规范里指出，在接收到正确的响应之前，最多有 64 个时钟脉冲（8 字节）。如果想克服这个限制，需要假定出现一个卡错误，并终止通信。

```
// now wait for a response up to 8 bytes delay
i = 9;
do {
    r = readSPI();       // check if ready
    if ( r != 0xFF) break;
} while ( --i > 0);

return ( r);
```

```
/* return response
    FF - timeout, no answer
    00 - command accepted
    01 - command received, card in idle state  (after RESET)
    other errors
*/

} // sendSDCmd
```

如果接收到响应码，每位的 1 都会表示一个可能的问题。

位 0 = 空闲

位 1 = 擦除重置

位 2 = 非法命令

位 3 = 通信 CRC 错误

位 4 = 擦除序列错误

位 5 = 地址错误

位 6 = 参数错误

位 7 = 总是 0

注意，执行 sendSDCmd() 函数后，SD 卡仍为选定（SDCS 低），因此诸如 Block Write 和 Block Read 等命令需要从卡中读取或者向卡中发送额外的数据才能继续。而在其他不需要额外数据传输的命令中，要记住在函数以后取消卡的选定（置 SDCS 为高电平）。此外，因为要将 SPI2 端口和其他外设共享（例如，Explorer16 板上的串行 EEPROM），就要确保 SD/MMC 卡在片选线（SDCS）的上升沿后还需要接收一些时钟周期（8 个周期就足够了）。根据 SD/MMC 的规范，存储卡可以完成一些重要的内部事务，包括正确释放 SDO 线，让其他设备在同一条总线上也能正确通信。

另一对宏也会一致地执行这个功能：

```
#define disableSD() SDCS = 1; clockSPI()
#define enableSD()  SDCS = 0
```

13.2.6 完成 SD/MMC 卡初始化

在存储卡有效应用到大容量存储之前，还需要完成一个命令定义序列。该序列在最初的 MMC 卡规范中就有定义，并且在 SD 卡规范中只做了很少的改动。因为这里并不打算使用 SD 卡标准的任何高级特性，所以只使用 MMC 的基本序列定义以赢得最大兼容性。该命令序列由以下 5 部分组成，如图 13-4 所示。

(1) 将存储卡插入连接器并通电。

(2) CS 先保持初始高电平（卡未选定）。

(3) 在卡可以接收命令之前，必须经过至少 74 个时钟脉冲。

(4) 然后选定卡，发送 RESET(CMD0) 命令：存储卡应该进入空闲状态（并激活 SPI 模式）。

(5) 发送 INIT(CMD1) 命令，一直执行，直到卡退出空闲状态。

下面是 initMedia() 函数的部分代码，将实现上面的 5 个步骤：

```
int initMedia( void)
{
    int i, r;


    // 1. while the card is not selected
    SDCS = 1;

    // 2. send 80 clock cycles to start up
    for ( i=0; i<10; i++)
        clockSPI();

    // 3. then select the card
    SDCS = 0;

    // 4. send a reset command to enter SPI mode
    r = sendSDCmd( RESET, 0); SDCS = 1;
    if ( r != 1)
        return 0x84;

    // 5. send repeatedly INIT
    i = 10000;              // allow for up to 0.3s before timeout
    do {
        r = sendSDCmd( INIT, 0); SDCS = 1;
        if ( r) break;
    } while( --i > 0);
    if (( i==0) || ( r!=1))
        return 0x85;        // timed out
```

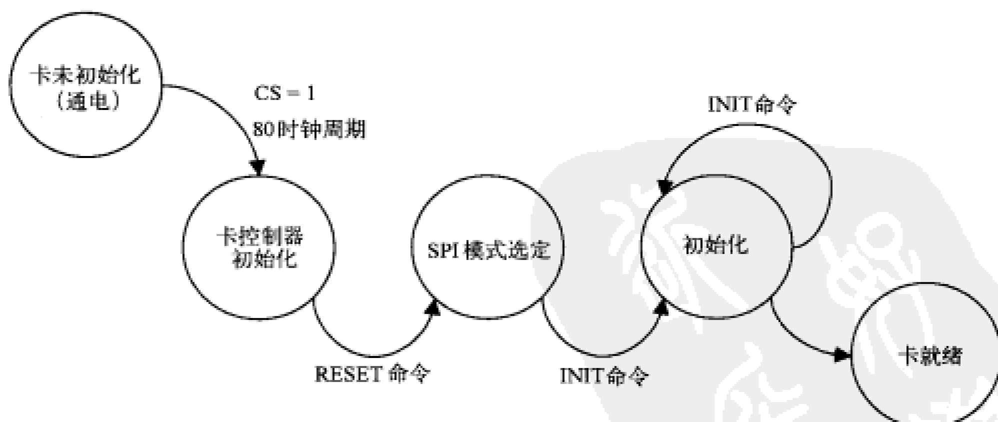


图 13-4　SD 卡初始化序列

根据不同存储卡的类型和大小，初始化命令需要一定的时间，通常是几十分之一秒。由于工作在 250Kb/s，因此每个字节的发送需要 32 μs。由于每个命令都需要 6 个字节，因此选择 10 000 的超时次数，就可以提供接近 2 s 的超时时间限制。

只有当上面的序列顺利完成时，才可以最终进入模式转换，并将时钟速度提升到硬件所支持的最大值。经过几次的试验，读者能够发现，由于设计合理的子板能提供 SD/MMC 连接器，

Explorer16 可以轻松地把时钟率保持在 8 MHz。这个值可以让 SPI 初级预分频器保持 1：1 的比率，二级预分频器保持 1：2 比率。添加下面的代码就可以完成 initMedia() 函数了：

```
    // 6. increase speed
    SPI2STAT = 0;           // disable momentarily the SPI2 module
    SPI2CON1 = 0x013b;      // change prescaler to 1:2
    SPI2STAT = 0x8000;      // re-enable the SPI2 module

    return 0;

} // init media
```

13.2.7　从 SD/MMC 卡读取数据

SD/MMC 卡是固态设备，有着典型的大型 Flash 存储器数组，因此用户可以在任何地址写入或读取任何大小的数据（在卡的存储范围以内）。实际上，出于对很多较早的大容量存储技术的兼容性考虑，在访问存储器的时候还是有一些限制的。所有的真实操作都定义在默认的 512 字节固定范围内。这与典型的个人电脑硬盘上的标准数据扇区大小也为 512 字节不谋而合。尽管可以通过一定的命令来改变这个大小，不过还是尽量保持默认的设置以保持兼容的优势。在后面的章节中，将通过一系列的子程序来完成与大多数普通 PC 操作系统兼容的文件系统。这样就可以通过个人电脑访问存储卡上的文件了，反之，个人电脑就可以访问应用写入的文件。

READ_SINGLE(CMD17)用于初始化指定存储地址上单扇区的传输。这个命令相当于一个 32 位"字节"地址变量，因此，为了避免混乱，在后面的内容中，统一只使用 LBA 或块地址，并且在向 READ_SINGLE 命令传递参数之前，必须先将真实字节地址的 LBA 乘以 512。

sendSDCmd() 函数是用于初始化读序列的（它会选定卡，并保持选定状态），然后当检查完响应码错误（应该是没有错误的）后，等待存储卡发送特殊令牌：DATA_START。它是数据块开始的唯一标记。这里再次强调，在初始化期间，无论时间多么充裕，都应该设置超时时间。因为在等待数据令牌期间只有 readSPI() 函数不停地被调用（每次只发送/接收一个字节），超时计数器值 10 000 可以提供大约 0.32s 的有效时间限制（相当充裕）。

一旦令牌被识别，就可以快速地读取指定数据块的全部 512 字节。数据后面是 16 位的 CRC 值，尽管没有用，但还是要读取的，如图 13-5 所示。

这样就可以取消存储卡的选定，终止整个读命令序列操作。

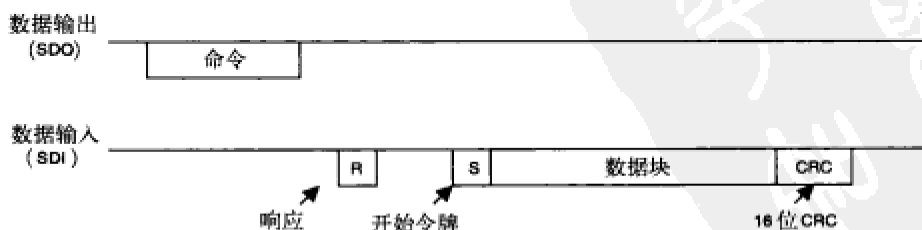


图 13-5　READ_SINGLE 命令期间的数据发送

子程序 readSECTOR()执行整个序列的几行代码如下：

```
// SD card responses
#define DATA_START                  0xFE

int readSECTOR( LBA a, char *p)
// a         LBA requested
// p         pointer to data buffer
// returns   TRUE if successful
{
    int r, i;

    READ_LED = 1;

    r = sendSDCmd( READ_SINGLE, ( a << 9));
    if ( r == 0)     // check if command was accepted
    {
        // wait for a response
        i = 10000;
        do{
            r = readSPI();
            if ( r == DATA_START) break;
        }while( --i>0);

        // if it did not timeout, read a 512 byte sector of data
        if ( i)
        {
            for( i=0; i<512; i++)
                *p++ = readSPI();

            // ignore CRC
            readSPI();
            readSPI();

        } // data arrived

    } // command accepted

    // remember to disable the card
    disableSD();
    READ_LED = 0;

    return ( r == DATA_START);      // return TRUE if successful
} // readSECTOR
```

为了给存储卡上的活动提供与硬盘和软盘驱动器上的活动相同的可视化指示，可以将 Explorer16 板上的一个 LED 作为“读”LED，以提醒用户不要在使用期间把卡拔掉。LED 会在每条读命令前打开，在完成的时候关闭。当然也可以有其他的形式。例如，就像其他的 USB Flash 设备，LED 可以在卡初始化的时候就打开，不用考虑是否有确实的命令在卡上运行。只有在调用了反初始化子程序的时候，LED 才关闭，提示用户可将卡移除。

数据输出 (SDO) 命令 S 数据块 CRC 开始令牌 16位CRC
数据输入 (SDI) R E 响应 数据接收 写操作完成

图 13-6 WRITE_SINGLE 命令期间的数据传输

13.2.8 向 SD/MMC 卡写入数据

基于与读函数相同的考虑，写函数的操作和“扇区”操作很相似——也是 512 字节数据。写序列将使用 WRITE_SINGLE 命令，不过这次的数据传输方向相反。当确定命令已被接受，则可以立刻发送 DATA_START 令牌，在 512 字节数据后，还跟随有两个字节用于 16 位的 CRC（任何的哑值就可以）。然后会暂停检查新的令牌：卡发送的 DATA_ACCEPT。它会证实整个数据块已经接收到，并且开始写操作。当卡在写入时，SDO 线会保持为低。在写命令完成之前，需要另一个循环来等待 SDO 线变回高电平。再次，必须使用超时时间来限制卡操作的完成时间。由于所有的 SD/MMC 存储卡都是基于 Flash 存储技术，因此写操作所需要的时间都会大于读操作的时间。超时值 10 000 能提供 0.3 s 的限制，这对于市场上最慢的存储卡也是足够了。

```
#define DATA_ACCEPT                 0x05

int writeSECTOR ( LBA a, char *p)
// a        LBA of sector requested
// p        pointer to sector buffer
// returns  TRUE if successful
{
    unsigned r, i;
    WRITE_LED = 1;

    r = sendSDCmd( WRITE_SINGLE, ( a << 9));
    if ( r == 0)     // check if command was accepted
    {
        writeSPI( DATA_START);
        for( i=0; i<512; i++)
            writeSPI( *p++);

        // send dummy CRC
        clockSPI();
        clockSPI();

        // check if data accepted
        if ( (r = readSPI() & 0xf) == DATA_ACCEPT)
        {
            for( i=10000; i>0; i--)
            {// wait for end of write operation
                if ( r = readSPI())
                    break;
            }
```

```
            } // accepted
            else
                r = FAIL;
        } // command accepted

        // to disable the card and return
        disableSD();
        WRITE_LED = 0;

        return ( r);        // return TRUE if successful

    } // writeSECTOR
```

类似于读程序，另一个 LED 应该用来表示写操作的执行并提醒用户。如果在卡写入期间拔出，数据很有可能丢失。

将目前的源代码保存在文件“sdmmc.c”中。

然后添加下面的两个函数，用于检测卡和写保护的位置：

```
int detectSD( void)
{
    return ( !SDCD);
} // detect SD

int detectWP( void)
{
    return ( !SDWP);
} // detect WP
```

> 注解　WP 转换只是起到提示作用，它并不是真正连接到硬件机制上阻止卡的写操作。检查 WP 的时间是由用户负责的。

最后，生成一个新的 include 文件“sdmmc.h”，来提供 SD/MMC 接口模块的原型和基本定义。

```
/*
** SD/MMC low level card interface
**
*/

#define TRUE    1
#define FALSE   0
#define FAIL    0

// IO definitions
#define READ_LED                    _RA1
#define WRITE_LED                   _RA2


typedef unsigned long LBA;      // logic block address, 32 bit wide


void initSD( void);
```

```
int  initMedia( void);

int detectSD( void);
int detectWP( void);

int readSECTOR  ( LBA, char *);
int writeSECTOR ( LBA, char *);
```

13.2.9 使用 SD/MMC 接口模块

无论读者信不信，到目前位置的六个小型的子程序已经满足了访问 SD/MMC 存储卡提供的所有永久性存储的需要。例如，512MB 卡提供了接近 1 000 000（是的，一百万）个独立的 512 字节的可寻址存储块（扇区）。目前，有如此容量的 SD/MMC 在美国零售价还不到 20 美元!

下面要生成一个小的测试程序来仿真 SD/MMC 模块的使用。其基本思想是仿真一个典型的应用，要求将大量的数据保存到 SD/MMC 存储卡上。数据有固定的块数，并写入预先设定的地址范围，然后通过读取数据来验证处理的成功完成。

打开一个新的源文件，添加常用的头文件，并且在 sdmmc.h 文件后加入处理器说明 include 文件。

```
/*
**  SDMMC read/write Test
**
*/

#include <p24fj128ga010.h>

#include "SDMMC.h"
```

然后定义两个字节数组，每个数组的大小默认是 SD/MMC 存储块的大小 512 字节。

```
#define B_SIZE           512            // sector/data block size
char     data[ B_SIZE];
char   buffer[ B_SIZE];
```

测试程序首先是加入一个指定的容易识别的图样，然后把内容重复写入存储卡。选定的地址范围由两个常量来定义：

```
#define START_ADDRESS       10000    // start block address
#define N_BLOCKS            1000     // number of blocks
```

Explorer16 演示板上 PORTA 的 LED 可为程序的正确执行和/或遇到错误提供视觉反馈。

使用主程序的前几行来初始化 SD/MMC 模块所需的 I/O 和连接到 LED 排的 POARTA 引脚。

```
main( void)
{
    LBA addr;
    int i, r;

    // I/O initializations
    TRISA = 0xff00;          // initialize PORTA LEDs output pins
    initSD();                // initialize all I/Os required for the SD/MMC module

    // fill the buffer with "data"
    for( i=0; i<B_SIZE; i++)
        data[i]= i;
```

下一个代码段可用来检查 SD 卡是否在插槽/连接器中。如果有需要，可以在循环中等待卡的检测，然后向合理的防反跳提供额外的延时。

```
// wait for card to be inserted
while( !detectSD());      // assumes SDCD pin is by default an input
Delayms( 100);            // wait for card contacts debounce and power up
```

防反跳延迟应该尽量长，以确保卡在开始“写入”命令前就已经安插稳当，否则有可能损坏卡中原有的数据。100 ms 的延时就足够了，Delayms()函数的实现可以使用 PIC24 定时器或者甚至是 RTCC 模块。下面是使用 Timer1 定时器模块的情形，假设处理器时钟是 32 MHz（这也正是 Explorer16 上的速度）。

```
void Delayms( unsigned t)
{
    T1CON = 0x8000;      // enable tmr1, Tcy, 1:1
    while (t--)
    {
        TMR1 = 0;
        while (TMR1<16000);
    }
} // Delayms
```

保持防反跳延迟函数同 detectSD()函数 SD/MMC 模块相分离，通常都是很重要的，因为它可以允许不同的应用，并且选择最佳的定时策略和优化资源分配。

一旦确定了卡的存在，就可以执行 initMedia()函数来进行初始化。

```
// initialize the memory card (returns 0 if successful)
r = initMedia();
if ( r)                    // could not initialize the card
{
    PORTA = r;             // show error code on LEDs
    while( 1);             // halt here
}
```

函数会返回一个整数，如果初始化序列顺利完成，返回的是 0，否则就是其他特定的错误码。在这个测试程序中，如果有任何初始化错误，那么只是在 LED 上显示错误码，然后停止执行，进入无限的循环。代码 0x84 和 0x85 分别表示 initMedia()函数的第四步和第五步出错了，也就是存储卡的 RESET 命令和 INIT 命令出错（失败或中止）。

如果一切正常，就可以处理真正的数据写入：

```
else
{
    // fill N_BLOCK blocks/SECTOR with the contents of data buffer
    addr = START_ADDRESS;
    for( i=0; i<N_BLOCKS; i++)
        if (!writeSECTOR( addr+i, data))
        {   // writing failed
            PORTA = 0x0f;
            while( 1);  // halt here
{
```

简单的 for 循环将从地址 10 000 到 10 999 的范围内重复执行 writeSECTOR()函数，不

停地复制相同的数据，并且验证每一个写命令是否都成功完成。如果出现了任何的写错误，那么 LED 上会显示一个特别的代码（0x0f），然后执行停止。实际上，相当于写入了一个 512 000 字节的文件。

```
// verify the contents of each block/SECTOR written
addr = START_ADDRESS;
for( i=0; i<N_BLOCKS; i++)
{   // read back one block at a time
    if (!readSECTOR( addr+i, buffer))
    {   // reading failed
        PORTA = 0xf0;
        while( 1);  // halt here
    }

    // verify each block content
    if ( !memcmp( data, buffer, B_SIZE))
    {   // mismatch
        PORTA = 0xff;
        while( 1); // halt here
    }
} // for each block
```

接下来，要开始新的循环，将每个数据块中的内容写入到第二个缓冲器，然后和第一个缓冲器中的原始图样进行比较。如果 readSECTOR() 函数失败，LED 上会显示错误码（0xf0）并停止测试。否则，标准的 C 库函数 memcmp() 会执行缓冲器内容的快速比较，如果两个数据如预期的那样完全相同，则返回 0。如果比较结果不匹配，那么就返回唯一的错误码（0x55）。为了访问标准 C 字符串库，需要添加新的 include 文件到列表中：

```
#include <string.h>
```

成功运行，完成主程序，点亮 PORTA 的 LED。

```
    } // else media initialized

    // indicate successful execution
    PORTA = 0xFF;
    // main loop
    while( 1);

} // main
```

如果读者已经在项目中加入了所需的源文件 “sdmmc.h”、“sdmmc.c” 和 “sdmmctest.c”，那么就可以使用标准列表来构建项目并且在 Explorer16 演示板上编程。正如在本章开始时说到的，读者还需要一个带有 SD/MMC 连接器的子板，才能真正地执行测试。不过，自制一块子板（或者购买一块子板）所付出的劳动将会从看到 PIC24 测试顺利执行时获得的喜悦那里得到回报。测试所需要的代码也是相当少的。

测试程序和 SD/MMC 访问模块一共只占用了处理器 Flash 程序存储器的 803 字（2 409 字节），小于总存储空间的 2%，如图 13-7 所示。和前面的章节一样，这个结果是关闭所有编译器优化选项时得到的。

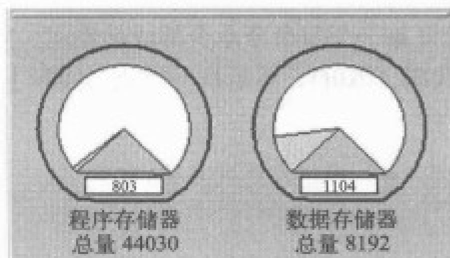


图 13-7 MPLAB 存储器使用情况窗口

13.3 飞后小结

根据作者的观点，没有比 SD/MMD 更便宜的大容量存储技术了。毕竟，只用到了很少的上拉电阻、便宜的连接器和一些 I/O 引脚，就可以大量地提升设备的存储能力。在 PIC24 所需的资源中，只用到了 SPI 外部模块，并且可以和其他外设共用。

方法的简便自然有其局限性。数据只能写入指定大小的块区，并且存储器数组中的位置也是有严格限定的。换而言之，不能与个人计算机或者其他能访问 SD/MMC 存储卡的设备共享数据，除非开发用户自定义应用。更糟的是，如果要使用一个已经被 PC 使用的卡，那么 PC 的数据就有可能被损坏，而整个数据卡可能需要重新格式化。在第 14 章中，将通过建立完整的文件系统库来处理这些问题。

13.4 提示与技巧

将默认块操作大小定为 512 字节，大部分是出于历史的原因。若将本章的低级访问子程序与其他大多数大容量存储媒介设备（包括硬盘驱动在内）的标准规格保持一致，那么开发下一层（文件系统）的操作将变得更加简单。但是如果要获得更大的性能，那这样做就是一个错误的选择了。实际上，如果要更快地写入，尤其要突破每种 Flash 存储媒介的瓶颈，最好选择更大的数据块。Flash 存储器通常提供很快的数据访问（读取），不过在写入时就变得相对慢了。写入需要两个步骤：第一是删除大量的数据（通常是整页），第二是在较小的块中执行真正的写入。存储数组越大，成比例地删除的页也会越大。例如，对于 512MB 存储卡，删除页会超过 2KB。由于这些细节对于用户来说是不可见的，卡内的主控制器负责删除/写序列和缓冲器，因此对于整个性能都会有影响。实际上，假设特定的 SD 卡有 2KB 的页，那么写入任意大小的数据（<2KB）则需要内部的卡控制器执行下面的步骤：

- ❑ 读取内部缓冲器中全部 2KB 块；
- ❑ 全部删除，等候删除时间；
- ❑ 缓冲器的部分位置写入新数据；
- ❑ 回写整个 2KB 块，等待写时间。

要在 512 字节大的块上执行写操作，写入 2KB 数据，函数库会要求 SD 卡控制器执行整个序列 4 次，而这个操作可以通过一个改变数据块长度或者使用多块写命令的序列来完成。尽管理论上，这个方法可以将前一个例子中的写入速度提高 400%，不过也应该考虑到这样的开销会相当高。实际上有以下这些缺点。

- 真正的存储器页大小可能是不可知的或者生产商未给出，尽管可以根据 Flash 媒介增加的密度来猜测（因此页大小也会增加）。
- PIC24 应用中 RAM 缓冲器的大小在增加，这是嵌入式程序的宝贵资源。
- 软件层次（将在第 14 章讨论）越高，不同大小的数据块就越难整合。
- 缓冲器越大，在卡移除时，丢失的数据会越多。

13.5 练习

(1) 通过测试不同的数据块大小，找出 SD 卡何时提供最好的写入性能。这样可以大概知道卡生产商提供的 Flash 存储器的实际页大小。

(2) 通过改变块长度来执行多块写命令，验证 SD 卡控制器的内部缓冲器是如何工作的，以及判断这两种方法是否等价。

13.6 推荐书目

- J. Axelson, 2006
  USB Mass Storage: Designing and Programming Devices and Embedded Hosts
  Lakeview Research, WI
  这本书延续了 Jan Axelson 优秀的 USB 系列书籍。正如在本章中看到的，SD/MMC 卡的低级接口是很简单的，不过大容量存储设备的 USB 接口是一个更加复杂的项目。

13.7 网上链接

- http://www.mmca.org/home
  多媒体卡协会（MMCA）的官方网站。
- http://www.sdcard.org/
  安全数码卡协会（SDCA）的官方网站。
- http://www.sdcard.org/sdio/Simplified%20SDIO%20Card%20Specification.pdf
  这是简化的 SDIO 卡规范。有了 SDIO，SD 接口就不只限于大容量存储了，它还可以成为很多先进的外设和装置的接口，如 GPS 接收器、数码相机等。

第 14 章　文件 I/O

本章内容

- ▶扇区和簇
- ▶文件分配表（FAT）
- ▶根目录
- ▶寻宝
- ▶打开文件
- ▶从文件读取数据
- ▶关闭文件
- ▶创建一个文件 I/O 模块
- ▶测试 fopenM()和 freadM()
- ▶向文件写数据
- ▶关闭文件，再次执行
- ▶附加功能
- ▶测试完整的文件 I/O 模块
- ▶代码规模

每次的飞行训练都会有一个由教练或者学校根据教学大纲制定的精确训练目标。本书每一章的飞行计划部分也都列出了该章所要达到的学习目标。但是真正的航空飞行计划是大不相同的，它包含了时间、纬度、航向、燃料消耗等所有的飞行数据。对于跨国飞行，飞行计划是一个非常重要的辅助手段，它可以帮助飞行员掌控局势，总能清楚飞机位置以及应急方案。认真编排飞行计划，呼叫飞行服务站（FSS），向空管员直接口述或通过互联网提交计划，都能获得额外的好处。一旦 FSS（和最终的 FAA）知道了飞机的地点、时间、飞行航线，就可以说，它们盯上飞机了。它们可以通过雷达（一种叫作飞行跟踪的设备）追踪到飞机，至少在飞机飞行高度太低以至无法跟踪的情况下，它们也能检测飞机是否在预计时间或者合理时间内到达目的地。如果没有收到飞行员的报告或者没有得到飞机飞抵目的机场的记录，就会立即展开搜救行动。尤其是在极端天气、山区或无人区的情况下，这种及时应对方案就是救命稻草。大多数飞行员在编排飞行计划的时候都感慨良多，感觉就像是一个少年告诉妈妈自己晚上活动的行踪，即使知道是为自己好，但也总是不喜欢这样做。和妈妈（也就是 FAA）共享信息，虽然需要伤点脑筋，但是却好处多多。

在嵌入式控制世界中，与 PC 机共享文件（信息）有诸多好处，但条件是必须掌握其中的规则——也就是说，必须知道 PC 机文件系统是如何工作的。

14.1　飞行计划

第 13 章开发了一个基本的接口模块（包含软件和硬件），用来访问 SD /MMC 卡并且支持需要大量数据存储的程序。对于其他的大容量存储设备，也可以使用类似的接口模块。但是本章内容主要集中在当大容量存储设备和一般的个人电脑操作系统（DOS、Windows 以及一些 Linux 系统）之间共享信息时所需要的算法和数据结构上。也就是说，本章将要开发一个模块用来访问标准文件系统（即通常所说的 FAT16）。第一个 FAT 文件系统是由比尔 · 盖茨（Bill

Gates）和马克·麦克唐纳（Marc McDonald）在 1977 年开发的，用于 Microsoft Disk BASIC 的磁盘管理。它采用的是多年以前已经用于文件系统的技术，并且在过去几十年中，不断涌现出众多的版本以适应容量和特性日益增多的大存储设备。在目前仍然使用的各个版本中，FAT12、FAT16 和 FAT32 是最常见的。特别是 FAT16 和 FAT32，可以被当前所有的 PC 操作系统所识别，并且根据效率和设备容量决定使用两者中的哪一种。最后，对于大多数的消费多媒体设备中的大容量 Flash，一般使用 FAT16 文件系统。

14.2 飞行

FAT 既是文件分配表（file allocation table）的首字母缩写，同时也是文件系统使用的一种很重要的数据结构的名称。不管怎样，文件系统是一种存储和组织电脑文件以及文件中数据的方法，通过文件系统可以使文件的查找和访问更容易。遗憾的是，在个人电脑的进程中，标准和技术是不断演化的结果，而不是进步的源头。基于这个原因，本书接下来将要介绍的 FAT 文件系统，只能说是尽量地与多年延续下来的大量技术和软件保持兼容性。

14.2.1 扇区和簇

FAT 文件系统的基本思想是很简单的。在第 13 章中，大多数大容量存储设备都遵循一个从硬盘技术中得到的“传统”：使用一个固定的 512 字节大小的块——“扇区”——来管理存储空间。在 FAT 文件系统中，有一小部分扇区预留作为总索引：文件分配表（FAT）。剩下的（大部分的）扇区用来存储数据，但并不是单独处理每一个扇区，而是将一串相邻的扇区连接起来形成一个新的、更大的存储单元，称为“簇”。簇可以小到只有一个扇区，也可以大到包含 64 个扇区（一般情况下）。文件分配表跟踪的是每个簇的使用情况及位置。因此，簇是 FAT 文件系统中真正意义上的最小存储单元。

如图 14-1 所示是一个简化的 FAT 结构文件系统的例子，该系统被格式化为 1 022 个簇，每个簇包含 16 个扇区。（注意：数据存储空间是从第二个簇开始的。）在本例中，每个簇可以存储 8KB 数据，因此整个文件系统的存储容量为 8MB。

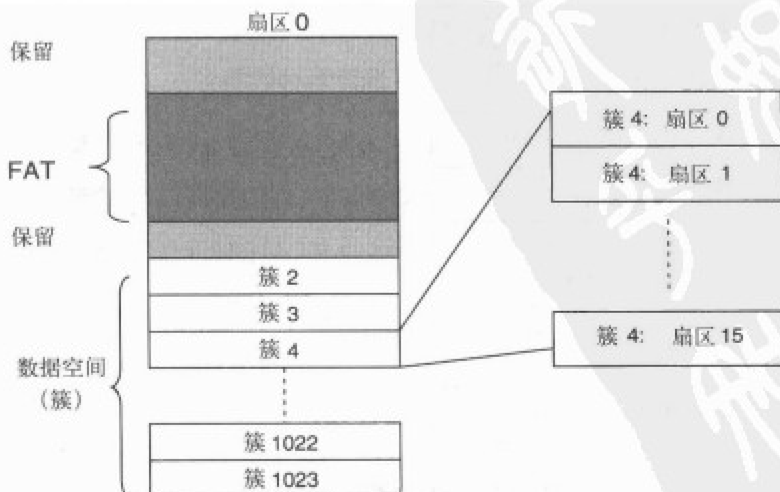


图 14-1 简化的 FAT 文件系统模型

注意：簇越大，需要用来管理整个存储空间的簇的个数就越少，文件分配表也就越小，因而文件系统的效率就越高。但是，当写入很多小的文件时，簇越大，浪费的空间就越多。在格式化一个使用 FAT 文件系统的存储设备时，操作系统的任务就是确定一个合适的簇大小以达到最佳的平衡。

14.2.2 文件分配表（FAT）

在 FAT16 文件系统中，文件分配表中每个簇都有一个 16 位整数值。如果一个簇是可用的并且是空的，则表中相应位置的值为 0x0000。如果一个簇处于使用状态并且包含了一个完整文件的数据，则表中相应位置的值为 0xFFFF。如果一个文件大于一个簇的大小，那么几个簇将会连接起来形成一个簇链。按顺序排列的每个簇都会包含链中下一个簇的序号。链中最后一个簇在文件分配表中相应位置的值就是 0xFFFF。此外，一些特定值用来标识保留簇（0x0001）和坏簇（0xFFF7）。由于 0x0000 和 0x0001 已被赋予了特殊意义，因此按照惯例，数据存储区从 2 号簇开始。相应地，FAT 中的前两个 16 位整数为保留值。

图 14-2 为前一个例子中提到的一个文件系统的 FAT（文件分配表）的内容。簇 0 和簇 1 为保留簇。簇 2 包含了一些数据，用来表示一个簇的部分或者全部（16 个）扇区，用来存储一个大小不超过 8KB 的文件的数据。

簇 3 是一个含有 3 个簇（簇 3、簇 4、簇 5）的链上的第一个簇。簇 3、簇 4 中所有的扇区，以及簇 5 中的部分或者全部扇区用来存储一个大小在 16KB 到 24KB 之间（目前为止只能这样假设）的文件数据。剩下的所有簇都为空的并且可用。

注意，一个 FAT 本身的大小是由所有的簇的个数乘以 2 得到的（每个簇为 2 字节），并且可以遍布多个扇区。在上个例子中，一个包含 1 024 个簇的文件系统的 FAT，需要 2 048 个字节，或者 4 个 512 字节的扇区。同样，由于文件分配表是整个 FAT 文件系统最关键的部分，因此在数据空间开始之前，需要保留连续的多个副本（通常是两个）。



图 14-2 文件分配表中的簇链

14.2.3 根目录

FAT 的任务是记录数据的存储方式以及位置。FAT 并不包含数据所属的文件的属性信息。为了得到文件属性，需要使用另外一种结构——根目录，根目录的任务是存储文件名、文件大小、日期、时间以及一些其他的属性。FAT16 文件系统为根目录（以下简称根）分配固定大小的空间和固定的位置——在 FAT（第二个副本）和第一个存储数据的簇之间，如图 14-3 所示。

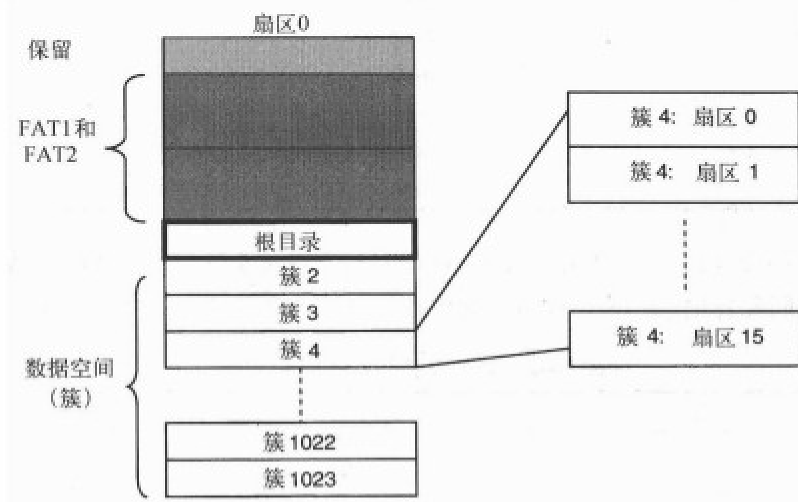


图 14-3　FAT 文件系统构架示例

由于位置和大小（包含的扇区的个数）都是固定的，因此一个根目录所能容纳的文件个数的上限（或目录实体个数）在格式化设备的时候就已经限定了。根目录中每个扇区允许录入 16 个文件条目，每个条目需要一块 32 字节大小的存储空间，如图 14-4 所示。

偏移量	字段	大小
偏移量：0	文件名	8 个 ASCII 码
偏移量：8	扩展名	3 个 ASCII 码
偏移量：11	属性	1 字节
	保留	
偏移量：22	时间	1 个字（16位）
偏移量：24	日期	1 个字（16位）
偏移量：26	第一个簇	1 个字（16位）
偏移量：28	文件大小	1 个 long 型数据（32位）

图 14-4　根目录条目的基本结构

对于文件名和扩展名，如果读者熟悉使用 8 : 3 约定（文件名和扩展名之间只需要用空格连接，点号可以丢弃）的旧版微软操作系统的话，将会发现文件名和扩展名这两个域是最明了的。

属性空间是由一组标志位组成的，其含义如表 14-1 所示。

表 14-1 目录条目中的文件属性

位	掩 码	描 述
0	0x01	只读
1	0x02	隐藏
2	0x04	系统
3	0x08	卷标
4	0x10	子目录
5	0x20	文档

时间项和日期项（如表 14-2 和表 14-3 所示）用来记录文件最近一次的修改时间和日期，同时还必须将时间和日期编码为一种特殊的格式，以将信息压缩在 2 个 16 位的字之内。

表 14-2 目录条目域中的时间编码

位	描 述
15–11	小时（0–23）
10–5	分钟（0–59）
4–0	秒钟（0–29）

表 14-3 目录条目域中的日期编码

位	描 述
15–9	年（0 = 1980，127 = 2107）
8–5	月（1 = 一月，12 = 十二月）
4–0	日（1–31）

注意，对于日期字段的编码，不允许将 0x0000 解释为合法日期。当该字段未被使用或者已被损坏时，有助于为发现文件系统提供线索。

第一个簇域为 FAT 提供了基本链接。该 16 位的字仅仅包含了存储整个文件数据所需要的簇（可能是簇链中唯一的一个或者第一个）的个数。

最后，大小域中包含的是文件数据以 long 整型（32 位）表示的字节大小。

通过观察一个目录条目中文件名的第一个字符，可以判断：若条目是正在使用的，则该字符为 ASCII 可打印字符；若条目为空，则第一个字节为 0，还可以推测出文件列表已经结束，因为文件系统是按顺序处理所有条目的。还有一种可能性：若文件从目录中被删除掉了，则文件名的第一个字符就会由一个特殊的码字（0xE5）代替。表明条目内容不再是有效的，下一次存储新文件时条目可以被重新使用。无论如何，在浏览列表寻找文件时，还要注意后续的条目。

14.2.4 寻宝

若要充分地了解一个 FAT16 文件系统的结构，还有很多需要学习。但是如果读者已经明白

了到目前为止的介绍，那么对整个结构的核心也会有一个合理的理解，继而准备寻求更多更具体的细节。因为要开始编写代码了。

迄今为止，本章所介绍的内容是简化后的，忽略了一些基础性的问题，比如以下问题。

- 从哪里了解一个存储设备的存储能力？
- 怎样知道 FAT 的位置？
- 怎样知道每个簇包含多少个扇区？
- 怎样知道数据空间是从哪里开始？

只要遵循一系列的步骤（就像小孩子的寻宝游戏一样），就会找到这些问题的答案。这里首先使用第 13 章的“sdmmc.c”模块函数中的 initSD() 函数来初始化 I/O 口，然后检查扩展槽中有没有存储卡：

```
// 0. init the I/Os
initSD();

// 1. check if the card is in the slot
if (!detectSD())
{
    FError = FE_NOT_PRESENT;
    return NULL;
}
```

然后，使用 initMedia() 函数继续初始化存储设备：

```
// 2. initialize the card
if ( initMedia())
{
    FError = FE_CANNOT_INIT;
    return NULL;
}
```

使用标准 C 语言库（stdlib.h）为两个数据结构动态分配空间：

```
// 3. allocate space for a MEDIA structure
D = (MEDIA *) malloc( sizeof( MEDIA));
if ( D == NULL)              // report an error
{
    FError = FE_MALLOC_FAILED;
    return NULL;
}

// 4. allocate space for a temp sector buffer
buffer = (unsigned char *) malloc( 512);
if ( buffer == NULL)         // report an error
{
    FError = FE_MALLOC_FAILED;
    free( D);
    return NULL;
}
```

第一个是名为 MEDIA 的结构体（稍后再详细介绍），这里有上面所有问题的答案（也许“宝

藏”这个名字更恰当一点)。

第二个结构体 buffer 只是一个 512 字节的数组，用来保存在“寻宝”过程中找到的数据扇区。

注意：若要使函数 malloc() 能够成功地分配存储区，读者必须记得为 Heap 预留部分 RAM 空间。提示：根据“Project Build” 列表学习怎样使用和修改项目 (project) 的连接器设置 (linker settings)。

主要是由于历史的原因，每个大容量存储设备的第一个扇区 (0 地址) 装载的都是主引导记录 (MBR)。

下面是通过第一次调用函数 readSECTOR()，访问主引导记录 (MBR)：

```
// 5.  get the Master Boot Record
if ( !readSECTOR( 0, buffer))
{
    FError = FE_CANNOT_READ_MBR;
    free( D); free( buffer);
    return NULL;
}
```

MBR 扇区的最后一个字为结束标志，由特殊值 0x55AA 组成，表示已经读取了正确的数据：

```
#define FO_SIGN        0x1FE  // MBR signature location (55,AA)


// 6. check if the MBR sector is valid
//      verify the signature word
if (( buffer[ FO_SIGN] != 0x55) ||
    ( buffer[ FO_SIGN +1] != 0xAA))
{
    FError = FE_INVALID_MBR;
    free( D); free( buffer);
    return NULL;
}
```

以前，该记录装载的是 PC 机系统上电时执行的代码。现在的个人电脑不再这样做了，况且 8 086 代码对于现在的 PIC24 程序是没有用处的。大多数时候，读者会发现除了一块以偏移量 0x1BE 开始的固定大小的位置之外，主引导记录是空的，大部分地方装载的是 0。在这里读者可以找到“分区表 (Partition Table)”，该表格只有 4 个条目，每个条目包含 16 字节。分区表在诸如 SD/MMC 等一些相对较小的存储卡上没有使用，但是由于兼容性的原因被保留下来，并且做成和现在使用的 PC 机上硬盘分区表完全一样的形式 (参见图 14-5)。

在程序里，安全的做法是假设整个存储卡只格式化一个分区，即分区表中的第一个也是唯一的一个条目 (16 字节块)。在这 16 字节中，只需要一小部分用来表示分区大小 (必须包括整个存储卡)、开始扇区，更重要的是其中包含的文件系统的类型。一些宏指令可以用来将缓冲区中的数据汇编成字和双字：

```
#define ReadW( a, f) *(unsigned *)(a+f)
#define ReadL( a, f) *(unsigned long *)(a+f)
```

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	03	....
000001C0	35	00	06	08	D8	C1	F1	00	00	00	0F	C9	0E	00	00	00	5...ØÁñ....É....
000001D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA	..............Uª

图 14-5 一个 MBR 扇区的十六进制数据

下面的定义用于指向 MBR 中正确的偏移量：

```
#define FO_FIRST_P    0x1BE  // offset of first partition table
#define FO_FIRST_TYPE 0x1C2  // offset of first partition type
#define FO_FIRST_SECT 0x1C6  // first sector of first partition offset
#define FO_FIRST_SIZE 0x1CA  // number of sectors in partition

    // 7. read the number of sectors in partition
    psize = ReadL( buffer, FO_FIRST_SIZE);

    // 8. check if the partition type is acceptable
    i = buffer[ FO_FIRST_TYPE];
    switch ( i)
    {
        case 0x04:
        case 0x06:
        case 0x0E:
            // valid FAT16 options
            break;
        default:
            FError = FE_PARTITION_TYPE;
            free( D); free( buffer);
            return NULL;
    } // switch
```

由于历史原因，一些 FAT16 文件系统使用的代码都能够被正确地译码，如 0x04、0x06 和 0x0E。

下面，为了将寻宝进行下去，拟从第一个分区条目中偏移量为 FO_FIRST_SECT 的地址中提取出一个双字（32 位）。

```
// 9. get the first partition first sector -> Boot Record
firsts = ReadL( buffer, FO_FIRST_SECT);
```

该扇区包含将要从设备中读取的下一个扇区的地址。

```
// 10. get the sector loaded (boot record)
if ( !readSECTOR( firsts, buffer))
{
    free( D); free( buffer);
    return NULL;
}
```

与主引导记录相似，标志字位于引导记录扇区的最后一个字中，在进行其他任务之前先检验这个标志字。

```
// 11. check if the boot record is valid
//       verify the signature word
if (( buffer[ FO_SIGN] != 0x55) ||
    ( buffer[ FO_SIGN +1] != 0xAA))
{
    FError = FE_INVALID_BR;
    free( D); free( buffer);
    return NULL;
}
```

以上叫做（第一分区的）引导记录（Root Record），其中包含了无用却是真正的可执行代码（如图 14-6 所示）。

幸运的是，在已知的固定位置上的相同记录里，有一直找寻的问题答案和有助于计算剩余位置和完成整个 FAT16 文件系统映射的其他元素。以下是引导记录缓冲区中重要的地址偏移量：

```
// Partition Boot Record key fields offsets
#define BR_SXC       0xd    // (byte) number of secotrs per cluster
#define BR_RES       0xe    // (word) number of reserved sectors for the boot record
#define BR_FAT_SIZE  0x16   // (word) FAT size in number of sectors
#define BR_FAT_CPY   0x10   // (byte) number of FAT copies
#define BR_MAX_ROOT  0x11   // (odd word) max number of entries in root dir
```

下面的代码可以用来计算出一个簇的大小：

```
// 12. determine the size of a cluster
D->sxc = buffer[ BR_SXC];
// this will also act as flag that the media is mounted
```

这确定了 FAT 的位置、大小和副本数目：

```
// 13. determine fat, root and data LBAs
// FAT = first sector in partition (boot record) + reserved records
D->fat = firsts + ReadW( buffer, BR_RES);
D->fatsize = ReadW( buffer, BR_FAT_SIZE);
D->fatcopy = buffer[ BR_FAT_CPY];
```

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0001E200	EB	00	90	20	20	20	20	20	20	20	20	00	02	20	01	00	▇.▌ .. ..
0001E210	02	00	02	00	00	F8	77	00	3F	00	10	00	F1	00	00	00	.....øw.?...ñ...
0001E220	0F	C9	0E	00	80	00	29	13	18	FD	E0	20	20	20	20	20	.É..▌.)..ýà
0001E230	20	20	20	20	20	20	46	41	54	31	36	20	20	20	00	00	FAT16 ..
0001E240	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E250	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E260	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E270	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E280	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E290	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E300	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E310	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E320	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E330	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E340	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E350	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E360	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E370	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E380	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E390	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA	..............Uª

图 14-6　一个引导记录的十六进制数据

而且确定根目录的位置：

```
// 14. ROOT = FAT + (sectors per FAT *  copies of FAT)
D->root = D->fat + ( D->fatsize * D->fatcopy);
```

值得注意的是，当已经准备好挖掘最后一片金子的时候，要时刻提防陷阱！

```
// 15. MAX ROOT is the maximum number of entries in the root directory
D->maxroot = ReadW( buffer, BR_MAX_ROOT) ;
```

看到了吗？没有？好吧，这里给出一个提示。观察前几行代码中定义的 BR_MAX_ROOT 地址偏移量，该地址是奇地址（0x11），而宏 ReadW()尝试将它作为双字节地址使用，从而导致进程陷阱并且重启 PIC24！

这里需要一种特殊的宏（可能相比之下效率稍低），通过每次只汇编一个字节来避免掉入陷阱中！

```
// these is the safe versions of ReadW to be used on odd address fields
#define ReadOddW( a, f) (*(a+f) + ( *(a+f+1) << 8))

// 15. MAX ROOT is the maximum number of entries in the root directory
D->maxroot = ReadOddW( buffer, BR_MAX_ROOT) ;
```

最后两条信息非常容易理解。通过它们，读者可以了解数据区（簇中的可用部分）从哪里开始，有多少个簇可以为程序所使用：

```
// 16. DATA = ROOT + (MAXIMUM ROOT *32 / 512)
D->data = D->root + ( D->maxroot >> 4); // assuming maxroot % 16 == 0!!!

// 17. max clusters in this partition = (tot sectors - sys sectors )/sxc
D->maxcls = (psize - (D->data - firsts)) / D->sxc;
```

尽管经过了多达 17 个小心谨慎的步骤才寻到宝藏，但已经找到了用来完全描述出 SD/SMMC 存储卡（或者其他任意一个大容量存储器）上 FAT16 文件系统结构图的所有信息。其实这个“宝藏”只不过是一个映射表，通过该映射表可以找到大容量存储器上的文件（如图 14-7 所示）。

下面将介绍整个 MEDIA 结构体的定义部分，它位于结构体的最开始的“堆（heap）”中，并且一直小心地装载。以下是保存“宝藏”的地方：

```
typedef struct {
    LBA      fat;              // lba of FAT
    LBA      root;             // lba of root directory
    LBA      data;             // lba of the data area
    unsigned maxroot;          // max number of entries in root dir
    unsigned maxcls;           // max number of clusters in partition
    unsigned fatsize;          // number of sectors
    unsigned char fatcopy;     // number of FAT copies
    unsigned char sxc;         // number of sectors per cluster
    } MEDIA;
```

现在可以将所有的步骤合成为一个 mount() 函数，类似于 Unix 系列操作系统的函数。

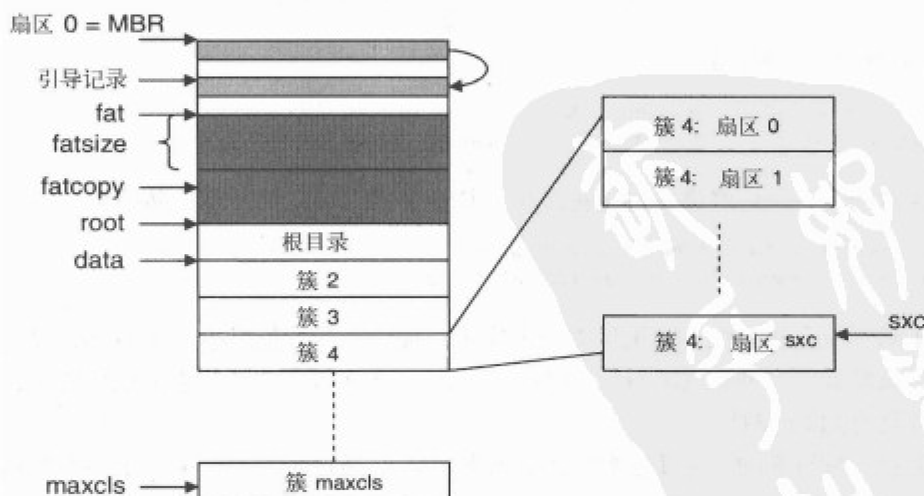


图 14-7　找到“宝藏”——FAT16 的完整结构图

对于一个使用于 Unix 系统的大容量存储器，设备上的文件系统必须被“贴装”，换而言之，就是作为主文件系统的一个新分支连接到系统上。Windows 用户可能对于这个概念还不是很熟

悉，因为不用选择什么时候或者在什么地方对新设备的文件系统进行贴装。在 Windows 系统上电或者插入可移动设备时，所有的大容量存储设备都是无条件地自动安装，并在 Windows 文件系统的根目录下为它们分配一个字符标识符（“C:”、“D:”、“E:”等）。

```
//-----------------------------------------------------------------
// mount    initializes a MEDIA structure for FILEIO access
//

MEDIA * mount( void)
{
    LBA  psize;       // number of sectors in partition
    LBA  firsts;     // LBA of first sector inside the first partition
    int i;
    unsigned char *buffer;

    ... insert here all 17 steps of our treasure hunt

    // 18. free up the temporary buffer
    free( buffer);
    return D;

} // mount
```

接下来再定义一个 MEDIA 结构（D）的全局指针，用来保存 mount() 函数返回的指针。它将作为整个文件系统的开始指针。起初，假设每次在给定的地方只有一个存储设备可用（如一个连接器/插槽，一个存储卡）。

```
// global definitions
MEDIA *D;
```

再定义一个函数 unmount()，它的任务仅仅是用来释放为 MEDIA 结构体分配的地址空间。

```
//-----------------------------------------------------------------
// unmount    releases the space allocated for the MEDIA structure
//

void unmount( void)
{
    free( D);
    D = NULL;
} // unmount
```

14.2.5 打开一个文件

在得到了存储设备的映射图之后，现在开始完成最初的目标：访问独立的文件。实际上，本章接下来所要介绍的内容是，一系列与大多数操作系统中用来进行文件操作的函数相类似的高级函数。这需要一个函数用于存储设备的文件地址搜索，一个函数用于按顺序读取文件数据，可能还需要一个函数写入数据和创建新文件。

按照逻辑顺序，下面从 fopenM() 函数开始开发。它的任务是找到关于文件（如果文件存在的话）的所有可能信息，同时将这些信息整合到一个新的结构体 MFILE 中。之所以使用 MFILE 作为该结构体的名字，是为了避免与标准 C 库“stdio.h”中已定义的相似的结构体或者函数

发生冲突。

```
typedef struct {
    MEDIA  * mda;               // MEDIA structure
    unsigned char * buffer;     // sector buffer
    unsigned cluster;           // first cluster
    unsigned ccls;              // current cluster in file
    unsigned sec;               // sector in current cluster
    unsigned pos;               // position in current sector
    unsigned top;               // number of data bytes in the buffer
    long     seek;              // position in the file
    long     size;              // file size
    unsigned time;              // last update time
    unsigned date;              // last update date
    char     name[11];          // file name
    char     chk;               // MFILE structure checksum = ~( entry + name[0])
    unsigned entry;             // entry position in cur directory
    char     mode;              // mode 'r', 'w'
    } MFILE;
```

乍看上去它好像很长——超过 40 字节——不过，当读者学习完本章的内容时，就会发现它们全部都是用得着的。从现在开始，读者要相信这一点。

模仿标准的 C 语言库实现（对于很多操作系统来说），fopenM()函数将接收两个（ASCII）字符串参数：文件名和“模式”字符串，包含“r”或者“w”，表明打开的文件是要读取还是要写入的。

```
MFILE *fopenM( const char *filename, const char *mode)
{
    char c;
    int i, r;
    unsigned char *b;           // newly allocated buffer
    MFILE *fp;                  // pointer to newly allocated MFILE structure
    MEDIA *mda=D;               // pointer to MEDIA structure
```

为了优化存储器的使用，MFILE 结构只在需要的时候分配。它实际上是 fopenM()函数的首要任务之一，数据结构的指针是它的返回值。为避免 fopenM()函数失败，将 NULL 指针用作错误报告。

当然，打开文件的先决条件是存储设备文件系统已经安排好，并且 mount()函数已经执行完毕。MEDIA 结构指针必须由全局 D 指针来放置。

```
// 1.  check if a storage device is mounted
if ( D == NULL)         // unmounted
{
    FError = FE_MEDIA_NOT_MNTD;
    return NULL;
}
```

由于存储设备的所有活动都必须在 512 字节的块中进行，因此需要分配同样大小的存储空间作为读/写缓冲区。

```
// 2. allocate a buffer for the file
b = (unsigned char*)malloc( 512);
if ( b == NULL)
{
    FError = FE_MALLOC_FAILED;
    return NULL;
}
```

只有在上述512字节的空间可用时，才能继续往下进行并为MFILE结构体分配更多的空间。

```
// 3. allocate a MFILE structure on the heap
fp = (MFILE *) malloc( sizeof( MFILE));
if ( fp == NULL)            // report an error
{
    FError = FE_MALLOC_FAILED;
    free( b);
    return NULL;
}
```

现在，缓冲区指针和MEDIA指针可以记录在MFILE结构体内了。

```
// 4. set pointers to the MEDIA structure and buffer
fp->mda = D;
fp->buffer = b;
```

文件名参数必须被提取出来，其中的每个字符都转换成大写字符（使用标准 C 库函数“ctype.h”），如果有需要的话，还应使用空格填补到8个字符长度。

```
// 5. format the filename into name
for( i=0; i<8; i++)
{
    c = toupper( *filename++);      // read a char and convert to upper case
    if (( c == '.') || ( c == '\0'))// extension or short name noextension
        break;
    else
        fp->name[i] = c;
} // for
// if short fill the rest up to 8 with spaces
while ( i<8) fp->name[i++] = ' ';
```

相似地，在删除点后，3个字符的扩展需要格式化和填补。

```
// 6. if there is an extension
if ( c != '\0')
{
    for( i=8; i<11; i++)
    {
        c = toupper( *filename++);      // read char, convert to upper case
        if ( c == '.')
            c = toupper( *filename++);
        if ( c == '\0')                 // short extension
            break;
        else
            fp->name[i] = c;
    } // for
    // if short fill the rest up to 3 with spaces
    while ( i<11) fp->name[i++] = ' ';
} // if
```

尽管大多数的 C 语言库提供多"模式"访问文件的扩展支持，例如区分文本和二进制文件，及提供"附加"选项，用户通常接受（最低限度）一个只有 2 个基本选项的子集："r"或者"w"。

```
// 7. copy the file mode character  (r, w)
if ((*mode == 'r')||(*mode == 'w'))
    fp->mode = *mode;
else
{
    FError = FE_INVALID_MODE;
    goto ExitOpen;
}
```

当文件名被正确格式化后，用户就可以开始从存储设备的根目录中搜寻有相同文件名的条目。

```
// 8. Search for the file in current directory
if ( ( r = findDIR( fp)) == FAIL)
{
    FError = FE_FIND_ERROR;
    goto ExitOpen;
}
```

现在可以抛开搜寻的细节，相信新的 findDIR() 函数会返回三个可能值：FAIL、NOT_FOUND 和最终的 FOUND。可能需要经常考虑出现的错误。毕竟，在考虑存储设备的主要致命错误的可能性前，总是有缺乏经验的用户会简单拔出存储卡。如果是那样的话，对于之前所有的错误，就不用其他的后续处理了。最好立即释放到目前为止已经分配的存储空间，同时，就像贴装过程一样，将一个错误代码放在专用"邮箱"FError 中，然后返回一个 NULL 指针。

但是，只要完成文件搜索而没有出现错误（不管错误有没有被找到），就可以继续初始化 MFILE 结构体了。

```
// 9. init all counters to the beginning of the file
fp->seek = 0;               // first byte in file
fp->sec = 0;                // first sector in the cluster
fp->pos = 0;                // first byte in sector/cluster
```

计数变量 seek 用来在顺序访问文件内容的时候跟踪所处文件中的当前位置。它的值是一个长整型（unsigned long），范围从 0 到整个文件系统以字节表示的数值大小。sec 用来跟踪当前正在操作的是哪个扇区（在当前簇内）。它的值是整数，范围从 0 到 sxc-1，即组成每个数据簇的扇区数目。最后，pos 用来跟踪将要访问的下一个字节（在当前缓冲区中），它的值是从 0 到 511 的整数。

```
// 10. depending on the mode (read or write)
if ( fp->mode == 'r')
{
```

这里，根据现有的文件是否需要打开读取或者是否需要新建一个文件用来写入，来决定需要做哪些事情。在读模式（"r"）下（这种情况下，文件更容易被找到）调用 fopenM() 函数时，需要先完成所有必要的步骤。

```
// 10.1 'r' open for reading
if  ( r == NOT_FOUND)
{
    FError = FE_FILE_NOT_FOUND;
    goto ExitOpen;
}
```

如果文件已找到，则使用函数 findDIR() 将填充 MFILE 结构体的更多域，包括以下几种。

- 条目（entry）：表示在根目录中找到文件的位置。
- 簇（cluster）：表示通过检索目录条目找到的存储文件数据的第一个数据簇的序号。
- 大小（size）：表示组成整个文件的字节数。
- 文件创建时间和日期。
- 文件属性（attributes）。

第一个簇将成为当前簇：ccls。

```
else
{  // found

// 10.2 set the current cluster pointer on the first file cluster
    fp->ccls = fp->cluster;
```

现在已经得到了识别缓冲区中第一个扇区所需的信息。函数 readDATA()（稍后再详细介绍）用来进行简单的计算，以将 ccls 和 sec 的值转换成数据区扇区的序号绝对值，同时使用低级函数 readSECTOR() 从存储设备中取出数据。

```
// 10.3 read a sector of data from the file
    if ( !readDATA( fp))
    {
        goto ExitOpen;
    }
```

注意文件的长度不需要强制为一个扇区大小的倍数。因此从缓冲区中读出的数据很有可能只有一部分是属于实际文件的。MFILE 结构的字段 top 用来记录实际的文件数据结束位置和可能插入的位置。

```
        // 10.4 determine how much data is really inside the buffer
            if ( fp->size-fp->seek < 512)

                fp->top = fp->size - fp->seek;

            else

                fp->top = 512;

    } // found
} // 'r'
```

由于这些是用来完成函数 fopenM()（当以读模式打开文件时）所真正需要的，因此现在可以返回指向 MFILE 结构体的指针。为了标识与指针使用和再使用相关的可能发生的错误，可以采取一个额外的安全措施，即计算校验和，只有整个文件成功打开时校验和才会是一个正确的结果。

```
    // 12.  compute the MFILE structure checksum
    fp->chk = ~( fp->entry + fp->name[0]);

    return fp;
```

注解　稍后将会在这个问题之前插入更多的代码，所以现在不用担心数字的顺序。

一旦前面有任何一个步骤失败了，将会退出函数，并且在释放分配给扇区缓冲的地址空间和分配给 MFILE 结构体的地址空间之后，返回 NULL 指针（空指针）。

```
        // 13. Exit with error
    ExitOpen:
        free( fp->buffer);
        free( fp);
        return NULL;

    } // fopenM
```

在自上向下模式中，现在可以完成 fopenM() 中使用的两个辅助函数，从 readDATA() 开始：

```
unsigned readDATA( MFILE *fp)
{
    LBA l;

    // calculate lba of cluster/sector
    l = fp->mda->data + (LBA)(fp->ccls-2) * fp->mda->sxc + fp->sec;

    return( readSECTOR( l, fp->buffer))

} // readDATA
```

注意怎样使用 MEDIA 结构体中的 data 和 sxc 来计算正确的扇区号。非常简单！

相似地，可以创建一个函数，用来从根目录中读取包含给定条目的一个数据块。

```
unsigned readDIR( MFILE *fp, unsigned e)
// loads current entry sector in file buffer
// returns      FAIL/TRUE
{
    LBA l;

    // load the root sector containing the DIR entry "e"
    l = fp->mda->root + (e >> 4);

    return ( readSECTOR( l, fp->buffer));

} // readDIR
```

由于每个目录的条目长度都是 32 字节，因此每个扇区包含 16 个条目。

在根目录下所有有效的条目中，函数 findDIR() 将作为封装在查找循环中的一系列步骤而被快速地编译。

```
unsigned findDIR( MFILE *fp)
// fp        file structure
// return    found/not_found/fail
{
    unsigned eCount;              // current entry counter
    unsigned e;                   // current entry offset in buffer
    int i, a, c, d;
    MEDIA *mda = fp->mda;

    // 1. start from the first entry
    eCount = 0;
    // load the first sector of root
    if ( !readDIR( fp, eCount))
        return FAIL;
```

现在开始装入第一个根目录扇区，该扇区在缓冲区中，包含最开始的 16 个条目。计算缓冲区中每个条目的地址偏移量：

```
// 2. loop until you reach the end or find the file
while ( 1)
{
// 2.0 determine the offset in current buffer
    e = (eCount & 0xf) * DIR_ESIZE;
```

检查文件名字条目的第一个字符：

```
// 2.1 read the first char of the file name
    a = fp->buffer[ e + DIR_NAME];
```

如果该值为 0，则表明是一个空的条目，同时也是列表的结尾。此时可以立即退出并报告文件名未找到。

```
// 2.2 terminate if it is empty (end of the list)
    if ( a == DIR_EMPTY)
    {
        return NOT_FOUND;
    } // empty entry
```

其他的可能是条目已标记为删除，这里可以忽略。

```
// 2.3 skip erased entries if looking for a match
    if ( a != DIR_DEL)
    {
```

否则，对于合法的条目，需要检查它的属性以确定它是否对应正确的文件或者其他任何类型的对象。可能是：子目录、卷标或者长文件名。这些都不是用户所关心的。尽量保持事情简单化，忽略新 FAT 文件系统标准中出现的高级和专有特性。

```
// 2.3.1 if not VOLume or DIR compare the names
    a = fp->buffer[ e + DIR_ATTRIB];

    if ( !((a & ATT_DIR) || ( a & ATT_VOL)) )
    {
```

逐字符地比较文件名，寻找一个完全匹配的项。

```
// compare file name and extension
for (i=DIR_NAME; i<DIR_ATTRIB; i++)
{
    if ( ( fp->buffer[ e + i]) != ( fp->name[i]))
        break;  // difference found
}
```

只有当每个字符都匹配时，才能从条目中提取必要的信息，并且将这些信息复制到 MFILE 结构体中，返回一个 FOUND（已找到）码。

```
            if ( i == DIR_ATTRIB)
            {
                // entry found, fill the mfile structure
                fp->entry = eCount;          // store entry index
                fp->time = ReadW( fp->buffer, e + DIR_TIME);
                fp->date = ReadW( fp->buffer, e + DIR_DATE);
                fp->size = ReadL( fp->buffer, e + DIR_SIZE);
                fp->cluster = ReadL( fp->buffer, e + DIR_CLST);
                return FOUND;
            }
        } // not a dir nor a vol
    } // not deleted
```

文件名和扩展名应该不同，下面将只是继续寻找下一个条目，记住每隔 16 个条目就要装载根目录的下一个扇区。

```
// 2.4 get the next entry
    eCount++;
    if ( eCount & 0xf == 0)
    {   // load a new sector from the Dir
        if ( !readDIR( fp, eCount))
            return FAIL;
    }
```

掌握了根目录（maxroot）所含条目的最大数目后，若在到达根目录的结尾时还没有表示 NOT_FOUND 的匹配，就需要结束搜索。

```
    //  2.5. exit the loop if reached the end or error
        if ( eCount >= mda->maxroot)
            return NOT_FOUND;             // last entry reached

    }// while

} // findDIR
```

14.2.6　从文件中读取数据

最终，期待已久的时刻到来了。文件系统已被贴装好，文件已找到并且被打开以供读取，为了从文件中自由地读取数据块，是时候创建函数 freeM() 了。

```
unsigned freadM( void * dest, unsigned size, MFILE *fp)
// fp       pointer to MFILE structure
```

```
// dest      pointer to destination buffer
// count     number of bytes to transfer
// returns   number of bytes actually transferred
{
    MEDIA * mda = fp->mda;  // media structure
    unsigned count=size;    // counts the number of bytes to be transferred
    unsigned len;
```

再次假设传递给该函数的参数名字、数量及参数顺序都与标准 C 语言库中的相似名字的函数一致。

目标缓冲器是指从文件中读取的数据被复制的地方，同时，传递一个正常的指针给开放的 MFILE 结构需要一定的字节数。

函数 freadM()从文件中读取尽可能多的所需字节，并且返回一个无符号整数值用以表示实际上已读取多少有效字节。在简单的操作中，如果返回值与调用程序要求的字节数不相同，则可以假定有重大故障发生。很多情况下，如果已经到达文件的结尾，只要不是其他类型的故障（比如在处理的过程中存储卡被拔出），就不需要被识别出来。

像往常一样，在成功打开一个文件的时候，不能相信参数中传递的指针，而要通过重新计算和比较由开放函数产生的校验和，来判断指针是不是指向 MFILE 结构。

```
// 1. check if fp points to a valid open file structure
if (( fp->entry + fp->name[0] != ~fp->chk ) || ( fp->mode != 'r'))
{   // checksum fails or not open in read mode
    FError = FE_INVALID_FILE;
    return size-count;
}
```

只有此时才能进入循环，从数据缓冲的扇区中开始传输数据。

```
// 2. loop to transfer the data
while ( count>0)
{
```

在循环中，第一个要检查的是当前位置（考虑到整个文件的大小)。

```
// 2.1 check if EOF reached
if ( fp->seek >= fp->size)
{
    FError = FE_EOF;  // reached the end
    break;
}
```

注意，这种错误只有在调用了 freadM()函数的应用忽略了过去的症状时才会产生，当最后调用的 freadM()返回的数据字节少于请求的数量时，或者当调用应用请求的字节数据与过去调用的文件中的可用数据完全相等时，则将出现这种错误。

否则，验证当前缓冲器中的数据是否已经被用完。

```
// 2.2 load a new sector if necessary
if (fp->pos == fp->top)
{
```

如有需要，重新设定缓冲区指针，并尝试着装入文件的下一个扇区：

```
        fp->pos = 0;
        fp->sec++;
```

如果当前簇中所有的扇区都已使用，可能不得不从 FAT 中按照簇链取出下一个簇：

```
        // 2.2.1 get a new cluster if necessary
        if ( fp->sec == mda->sxc)
        {
            fp->sec = 0;
            if ( !nextFAT( fp, 1))
            {
                break;
            }
        }
```

另一种情况是，向缓冲区放入新的数据扇区，注意检查该扇区是否是文件的最后一个扇区或者它只有一部分被填充数据：

```
        // 2.2.2 load a sector of data
        if ( !readDATA( fp))
        {
            break;
        }
        // 2.2.3 determine how much data is really inside the buffer
        if ( fp->size-fp->seek < 512)
            fp->top = fp->size - fp->seek;
        else
            fp->top = 512;
    } //  load new sector
```

在确定了缓冲器中存在数据且已准备好传输之后，再确定每块可以传输多少数据：

```
    // 2.3 copy as many bytes as possible in a single chunk
    // take as much as fits in the current sector
    if ( fp->pos+count < fp->top)
        len = count;                    // fits all in current sector
    else
        len = fp->top - fp->pos;        // take a first chunk, there is more

    memcpy( dest, fp->buffer + fp->pos, len);
```

使用标准 C 语言函数库（string.h）中的函数 memcpy() 来将一个数据块从文件缓冲器复制到目标缓冲器，同时由于这些程序是基于执行速度优化的，因此性能将会很好。指针和计数值不断地更新，循环不停地重复，直到所有请求的数据都传送完毕。

```
        // 2.4 update all counters and pointers
        count-= len;            // compute what is left
        dest += len;            // advance destination pointer
        fp->pos += len;         // advance the pointer in current sector
        fp->seek += len;        // advance the seek pointer

    } // while count
```

最后，退出函数，返回循环中实际传送的字节数目：

```
        // 3. return number of bytes actually transferred
        return size-count;

    } // freadM
```

14.2.7 关闭一个文件

由于只能打开文件来读取（使用目前为止已经定义的 freadM() 函数），因此关闭文件并不需要很多工作。可以考虑使函数 fopenM() 产生的检验和变为无效，并释放为 MFILE 结构和扇区缓冲分配的所有地址空间。

```
unsigned fcloseM( MFILE *fp)
{
    // 1. invalidate the file structure
    fp->chk = fp->entry + fp->name[0];      // set checksum invalid!

    // 2. free up the buffer and the MFILE struct
    free( fp->buffer);
    free( fp);

} // fcloseM
```

14.2.8 创建文件 I/O 模块

创建一个小型的库模块，将目前已经写完的所有函数都保存在一个函数中，命名为"fileio.c"。在头文件中加入一些包含文件。

```
/*
**  FILE I/O interface
**
** module: fileio.c
**
*/

// standard C libraries used
#include <stdlib.h>       // NULL, malloc, free...
#include <ctype.h>        // toupper...
#include <string.h>       // memcpy...

#include "sdmmc.h"        // sd/mmc card interface
#include "fileio.h"       // file I/O routines
```

当然，还需要再创建一个文件"fileio.h"，使之包含所有在后面的程序中可能用到的定义和原型。

```
/*
** FILE I/O interface
**
** FAT16 support
**
** module:    fileio.h
*/

extern char FError;   // mailbox for error reporting

// FILEIO ERROR CODES
#define FE_IDE_ERROR          1   // IDE command execution error
#define FE_NOT_PRESENT        2   // CARD not present
#define FE_PARTITION_TYPE     3   // WRONG partition type, not FAT12
```

```
#define FE_INVALID_MBR        4   // MBR sector invalid signature
#define FE_INVALID_BR         5   // Boot Record invalid signature
#define FE_MEDIA_NOT_MNTD     6   // Media not mounted
#define FE_FILE_NOT_FOUND     7   // File not found in open for read
#define FE_INVALID_FILE       8   // File not open
#define FE_FAT_EOF            9   // Fat attempt to read beyond EOF
#define FE_EOF               10   // Reached the end of file
#define FE_INVALID_CLUSTER   11   // Invalid cluster value > maxcls
#define FE_DIR_FULL          12   // All root dir entry are taken
#define FE_MEDIA_FULL        13   // All clusters in partition are taken
#define FE_FILE_OVERWRITE    14   // A file with same name exists already
#define FE_CANNOT_INIT       15   // Cannot init the CARD
#define FE_CANNOT_READ_MBR   16   // Cannot read the MBR
#define FE_MALLOC_FAILED     17   // Malloc could not allocate the MFILE struct
#define FE_INVALID_MODE      18   // Mode was not r.w.
#define FE_FIND_ERROR        19   // Failure during FILE search

typedef struct {
    LBA      fat;                 // lba of FAT
    LBA      root;                // lba of root directory
    LBA      data;                // lba of the data area
    unsigned maxroot;             // max number of entries in root dir
    unsigned maxcls;              // max number of clusters in partition
    unsigned fatsize;             // number of sectors
    unsigned char fatcopy;        // number of copies
    unsigned char sxc;            // number of sectors per cluster (!=0 flags media mounted)
    } MEDIA;


typedef struct {
    MEDIA * mda;                  // media structure pointer
    unsigned char * buffer;       // sector buffer
    unsigned cluster;             // first cluster
    unsigned ccls;                // current cluster in file
    unsigned sec;                 // sector in current cluster
    unsigned pos;                 // position in current sector
    unsigned top;                 // number of data bytes in the buffer
    long     seek;                // position in the file
    long     size;                // file size
    unsigned time;                // last update time
    unsigned date;                // last update date
    char     name[11];            // file name
    char     chk;                 // checksum = ~( entry + name[0])
    unsigned entry;               // entry position in cur directory
    char     mode;                // mode 'r', 'w', 'a'
    } MFILE;

// file attributes
#define ATT_RO      1             // attribute read only
#define ATT_HIDE    2             // attribute hidden
#define ATT_SYS     4             //  "      system file
#define ATT_VOL     8             //  "      volume label
#define ATT_DIR     0x10          //  "      sub-directory
#define ATT_ARC     0x20          //  "      (to) archive
#define ATT_LFN     0x0f          // mask for Long File Name records
```

```
#define FOUND        2           // directory entry match
#define NOT_FOUND    1           // directory entry not found


// macros to extract words and longs from a byte array
// watch out, a processor trap will be generated if the address is not word
aligned
#define ReadW( a, f) *(unsigned *)(a+f)
#define ReadL( a, f) *(unsigned long *)(a+f)

// this is a "safe" version of ReadW to be used on odd address fields
#define ReadOddW( a, f)  (*(a+f) + ( *(a+f+1) << 8))


// prototypes
unsigned nextFAT( MFILE * fp, unsigned n);
unsigned newFAT(  MFILE * fp);

unsigned readDIR( MFILE *fp, unsigned entry);
unsigned findDIR( MFILE *fp);
unsigned newDIR ( MFILE *fp);

MEDIA * mount( void);
void    unmount( void);

MFILE *  fopenM  ( const char *name, const char *mode);
unsigned freadM  ( void * dest, unsigned count, MFILE *);
unsigned fwriteM ( void * src, unsigned count, MFILE *);
unsigned fcloseM ( MFILE *fp);
```

不用担心，可能读者暂时并不能够掌握这些函数，本章的剩余部分将会继续向读者介绍它们。

14.2.9 测试 fopenM()和 fcloseM()

自上次开发项目以来，已经好久没有建立新项目了。为了验证已经编写好的代码，必须了解一个重要的部分——线程内核，没有它，程序将无法工作。由于现在已经有了线程内核功能，那么可以第一次编写一个小的测试程序，用来从 SD/MMC 卡（FAT 文件系统）上读取文件。

思路如下：将一个文本文件（任何文本文件都可以）从 PC 机上复制到 SD/MMS 卡上，然后让 PIC24 使用新的模块“fileio.c”读取文件，并将内容发送到 PC 机的串行口（超级终端或者任何一种使用 RS232 串行口的终端或打印机）。

以下是将要保存为“ReadTest.c”的主要模块：

```
/*
**      ReadTest.c
**
*/

#include <p24fj128ga010.h>

#include "SDMMC.h"
#include "fileio.h"
#include "../delay/delay.h"
```

```
#include "../3 comm/conu2.h"


#define B_SIZE  10
char data[ B_SIZE];

main( void)
{
    MFILE *fs;
    unsigned i, r;

    //initializations
    initU2();                   //115,200 baud 8,n,1

    putsU2( "init");

    while( !detectSD());        // assumes SDCD pin is by default an input
    Delayms( 100);              // wait for card to power up

    putsU2("media detected");

    if ( mount())
    {
        putsU2( "mount");
        if ( fs = fopenM( "name.txt", "r"))
        {
            putsU2("file opened");
            do{
                r = freadM( data, B_SIZE, fs);
                for( i=0; i<r; i++)
                    putU2( data[i]);
            } while( r==B_SIZE);
            fcloseM( fs);
            putsU2("file closed");
        }
        else
            putsU2("could not open file");

        unmount();
        putsU2("media unmounted");
    }

    // main loop
    while( 1);
} // main
```

下面将用到前几章创建的串行通信模块“conu2.c”和延时模块，该延时模块提供的函数 delayms()与前面用来测试模块“sdmmc.c”的函数很相似。它们的操作顺序也基本相同，只是这次并不是先调用函数 initMedia()，然后直接对 SD/MMC 卡的扇区进行读操作和写操作，而是通过调用函数 mount()来访问存储卡上的 FAT16 文件系统。这里将会使用“适当的”文件名来打开数据文件，以任意长度的数据块为单位从文件中读取数据，并且将文件内容送到 Explorer16 板上的串行口。

读完整个文件的内容之后，关闭文件，释放所有已使用的存储空间。

创建一个新的项目之后，把所有需要的模块添加到项目窗口中，它们包括：

- "sdmmc.c"；
- "fileio.c"；
- "conu2.c"；
- "delay.c"；
- "readtest.c"。

以及所有相应的 include 文件（.h）。

记住，要遵循新项目和 ICD2 调试程序的列表，这样才不会忘记为连接器设置 ICD2 选项。在同一个设置对话框中，别忘了为堆添加一些空间，这样才能保证为文件系统结构和缓冲器动态分配存储空间。

构建项目，并对 Explorer16 板进行编程，开始运行测试程序。

如果顺利的话，读者可以看到文件内容在终端机的屏幕上滚动，除了文件的最后部分，其他的内容有可能因滚动太快而不容易阅读。

注意，读者可以重新编译这个项目，使用不同大小的数据缓冲（从一个字节到 PIC24 所允许的最大存储空间的大小）运行测试程序。只要文件中有数据，读者要求读取多少扇区的数据，函数 freadM()就可以读取多少。

14.2.10 向文件写入数据

离结束还早着呢。如果模块"fileio.c"没有创建新文件的功能，那么它就是不完整的。这就需要读者创建一个函数 fwriteM()，并且完成函数 fopenM()。实际上到目前为止，当文件无法找到或者模式不是"读"模式的时候，函数 fopenM()会返回一个错误码。而这正是为了写入而打开文件所需要的。在检查模式参数值的时候，需要添加一个新的选项。此时，在第一次扫描想要访问的目录期间，文件的状态是 NOT_FOUND。

```
else // 11.  open for 'write'
  {
      if  ( r == NOT_FOUND)
      {
```

新文件需要分配一个新的簇来装载数据。使用函数 newFAT()在 FAT 中搜寻被标记（使用 0x0000）为可用的簇。搜索也许会失败，在函数可能返回的很多参数中，将会有一个错误报告用来说明存储空间已满，而且所有的簇都被占用。如果搜索成功，则记下这个新的簇的位置，更新 MFILE 结构，使之成为新文件的第一个簇。

```
// 11.1 allocate a first cluster to it
fp->ccls = 0;                          // indicate brand new file
if ( newFAT( fp) == FAIL)
{ // must be media full
    FError = FE_MEDIA_FULL;
    goto ExitOpen;
}
fp->cluster = fp->ccls;
```

接下来要在目录中为新文件寻找一个可用的条目空间。这需要再次用到根目录，寻找被标

记为已删除（代号 0xE5）的条目中的第一个条目，或者是列表结尾的空条目（标记为代号 0x00）中的第一个。

```
// 11.2 create a new entry
// search again, for an empty entry this time
if ( (r = newDIR( fp)) == FAIL)          // report any error
{
    FError = FE_IDE_ERROR;
    goto ExitOpen;
}
```

函数 newDIR() 负责查找一个可用的空条目，而且与之前使用过的函数 findDIR() 相似，都会返回以下 3 种可能的代码。

- FAIL，表明出现了大问题（或者是存储卡被拔掉了）。
- NOT_FOUND，表明根目录已满。
- FOUND，表明可用条目已被识别。

```
// 11.3 new entry not found
if ( r == NOT_FOUND)
{
    FError = FE_DIR_FULL;
    goto ExitOpen;
}
```

前两种情况必须报告错误，而且无法继续执行下去。但是如果条目已找到，那么就需要很多的工作来初始化该条目。

计算条目在当前缓冲器中的地址偏移量，然后用 MFILE 结构中的数据来装填该条目域。首先是文件大小：

```
else // 11.4 new entry identified fp->entry filled
{
    // 11.4.1 init file size
    fp->size = 0;

    // 11.4.2    determine offset in DIR sector
    e = (fp->entry & 0xf) * DIR_ESIZE;     // 16 entry per sector

    // 11.4.3 set initial file size to 0
    fp->buffer[ e + DIR_SIZE]   = 0;
    fp->buffer[ e + DIR_SIZE+1]= 0;
    fp->buffer[ e + DIR_SIZE+2]= 0;
    fp->buffer[ e + DIR_SIZE+3]= 0;
```

时间域和日期域可以从 RTCC 模块寄存器或者其他任何一个程序可用的计时结构中得到，而这里提供的默认值只是用来示范的。

```
fp->date = 0x34FE; // July 30th, 2006
fp->buffer[ e + DIR_DATE]   = fp->date;
fp->buffer[ e + DIR_DATE+1]= fp->date>>8;
fp->buffer[ e + DIR_TIME]   = fp->time;
fp->buffer[ e + DIR_TIME+1]= fp->time>>8;
```

完成一个目录条目还需要文件第一个簇的序号、文件名以及其他的属性（默认值）：

```
            // 11.4.5 set first cluster
            fp->buffer[ e + DIR_CLST]   = fp->cluster;
            fp->buffer[ e + DIR_CLST+1]= (fp->cluster>>8);

            // 11.4.6 set name
            for ( i = 0; i<DIR_ATTRIB; i++)
                fp->buffer[ e + i] = fp->name[i];

            // 11.4.7 set attrib
            fp->buffer[ e + DIR_ATTRIB] = ATT_ARC;

            // 11.4.8  update the directory sector;
            if ( !writeDIR( fp, fp->entry))
            {
                FError = FE_IDE_ERROR;
                goto ExitOpen;
            }
        } // new entry
    } // not found
```

回到第一次搜索整个根目录后的结果——万一真的找到了同样名字的文件，则需要报告错误。

```
    else // file exist already, report error
    {
        FError = FE_FILE_OVERWRITE;
        goto ExitOpen
    }
```

还有一种方法，就是首先删除当前条目，释放所有已使用的簇，然后从头开始。毕竟，将问题报告成一个错误是一个摆脱困境的更简单方法。

函数 fopenM() 所要求的修改就介绍到这里。在仿照标准 C 语言函数库中的一个命名类似的函数后，开始编写合适的新函数 fwriteM()。

```
unsigned fwriteM(  void *src, unsigned count, MFILE * fp)
// src       points to source data (buffer)
// count     number of bytes to write
// returns   number of bytes actually written
{
    MEDIA *mda = fp->mda;
    unsigned len, size = count;

// 1.  check if file is open
if ( fp->entry + fp->name[0] != ~fp->chk )
{   // checksum fails
    FError = FE_INVALID_FILE;
    return FAIL;
}
```

该函数的传递参数与函数 freadM() 使用的一致，同时对 MFILE 结构（当作参数传递）的整体性进行的第一个测试也是如此。

函数的核心也是一个循环：

```
// 2. loop writing count bytes
while ( count>0)
{
```

为了让每次传递的数据尽可能地多，我们使用“string.h”函数库中的快速函数memcpy()。

```
// 2.1 copy as many bytes at a time as possible
if ( fp->pos+count < 512)
    len = count;
else
    len = 512- fp->pos ;

memcpy( fp->buffer+ fp->pos, src, len);
```

在向缓冲器添加数据和增加文件大小的时候，为了记录当前的位置，需要更新大量的指针和计数器。

```
// 2.2 update all pointers and counters
fp->pos+=len;        // advance buffer position
fp->seek+=len;       // count the added bytes
count-=len;          // update the counter
src+=len;            // advance the source pointer

// 2.3 update the file size too
if (fp->seek > fp->size)
    fp->size = fp->seek;
```

一旦缓冲区满，则要将缓冲区的数据传送到存储卡上当前簇内的一个扇区中。

```
// 2.4 if buffer full, write current buffer to current sector
if (fp->pos == 512)
{
    // 2.4.1 write buffer full of data
    if ( !writeDATA( fp))
        return FAIL;
```

注意，如果这时候出现错误，那么将会是致命的。此时会返回一个代码FAIL，其值为0，表示一个数据都没有传输；而实际上是，到目前为止所有被写入存储卡中的数据全部丢失了。

如果所有的步骤都是正确的，那么将扇区指针增1。如果当前簇中的所有扇区都使用完了，则需要考虑再分配一个新的簇，再次命名为newFAT()。

```
        // 2.4.2 advance to next sector in cluster
        fp->pos = 0;
        fp->sec++;

        // 2.4.3 get a new cluster if necessary
        if ( fp->sec == mda->sxc)
        {
            fp->sec = 0;
            if ( newFAT( fp)== FAIL)
                return FAIL;
        }
    } //  store sector

} // while count
```

在开发函数 newFAT()的时候，必须确保该函数在簇链被添加到文件中时能精确地维持簇链在 FAT 中顺序不变。

```
    // 3.  number of bytes actually written

    return size-count;

} // fwriteM
```

函数已经完成，现在可以报告退出循环时实际写入的字节数。

14.2.11 关闭文件，第二次执行

关闭已打开的用于读取的文件仅仅是释放堆中存储空间的一种简单的形式和方法。而关闭一个已打开的用于写入的文件，则需要进行相当多的操作。

现在需要一个新的、改进的函数 fcloseM()，并且从模式域的检查开始。

```
unsigned fcloseM( MFILE *fp)
{
    unsigned e, r;              // offset of directory entry in current buffer

    r = FAIL;

    // 1. check if it was open for write
    if ( fp->mode == 'w')
    {
```

事实上，在关闭一个文件的时候，缓冲器中可能还会存在一些数据需要被写入设备中（即使这些数据不足一个扇区）。

```
// 1.1 if the current buffer contains data, flush it
if ( fp->pos >0)
{
    if ( !writeDATA( fp))
        goto ExitClose;
}
```

再重申一次，这个时候发生的任何错误都是致命的。它意味着由于 fcloseM()函数没有完成，文件的所有数据都丢失了。

检索合适的根目录扇区，计算缓冲区中目录条目的地址偏移量。

```
// 1.2      finally update the dir entry,
// 1.2.1    retrive the dir sector
if ( !readDIR( fp, fp->entry))
    goto ExitClose;

// 1.2.2    determine position in DIR sector
e = (fp->entry & 0xf) * DIR_ESIZE;    // 16 entry per sector
```

接下来，使用实际的文件大小（文件大小之前被初始化为 0）更新根目录中的文件条目的内容。

```
// 1.2.3 update file size
fp->buffer[ e + DIR_SIZE]   = fp->size;
fp->buffer[ e + DIR_SIZE+1]= fp->size>>8;
fp->buffer[ e + DIR_SIZE+2]= fp->size>>16;
fp->buffer[ e + DIR_SIZE+3]= fp->size>>24;
```

最终，包含条目内容的整个根目录扇区被重新写回到存储媒介中。

```
    // 1.2.4    update the directory sector;
    if ( !writeDIR( fp, fp->entry))
        goto ExitClose;
} // write
```

如果进行顺利的话，完成函数 fcloseM()使检验和的域无效，以防止在偶然情况下 MFILE 结构的重复使用，并且释放已经使用的存储空间和缓冲区。

```
    // 2. exit with success
    r = TRUE;

ExitClose:
    // 3. invalidate the file structure
    fp->chk = fp->entry + fp->name[0];      // set checksum wrong!

    // 4. free up the buffer and the MFILE struct
    free( fp->buffer);
    free( fp);

    return( r);

} // fcloseM
```

14.2.12 辅助函数

在完善函数 fopenM()、fcloseM()和创建新的函数 fwriteM()的时候，使用了一些低级函数执行重要的重复操作。

从函数 newDIR()开始。该函数用于查找根目录中可以新建文件的可用位置。虽然与函数 findDIR()有明显的相似之处，但是所执行的任务却是完全不同的。

```
unsigned newDIR( MFILE *fp)
// fp        file structure
// return    found/fail,  fp->entry filled
{
    unsigned eCount;              // current entry counter
    unsigned e;                   // current entry offset in buffer
    int a;
    MEDIA *mda = fp->mda;

    // 1. start from the first entry
    eCount = 0;
    // load the first sector of root
    if ( !readDIR( fp, eCount))
        return FAIL;

    // 2. loop until you reach the end or find the file
```

```
    while ( 1)
    {
    // 2.0 determine the offset in current buffer
        e = (eCount&0xf) * DIR_ESIZE;

    // 2.1 read the first char of the file name
        a = fp->buffer[ e + DIR_NAME];

    // 2.2 terminate if it is empty (end of the list) or deleted
        if (( a == DIR_EMPTY) ||( a == DIR_DEL))
        {
            fp->entry = eCount;
            return FOUND;
        } // empty or deleted entry found

    // 2.3 get the next entry
        eCount++;
        if ( (eCount & 0xf) == 0)
        { // load a new sector from the root
            if ( !readDIR( fp, eCount))
                return FAIL;
        }

    // 2.4 exit the loop if reached the end or error
        if ( eCount > mda->maxroot)
            return NOT_FOUND;                // last entry reached
    }// while

    return FAIL;
} // newDIR
```

使用函数 newFAT()寻找一个可用的簇，分配给新的数据块或者新文件。

```
unsigned newFAT( MFILE * fp)
// fp          file structure
// fp->ccls        ==0 if first cluster to be allocated
//                 !=0 if additional cluster
// return      TRUE/FAIL
//  fp->ccls new cluster number
{
    unsigned i, c = fp->ccls;

    // sequentially scan through the FAT looking for an empty cluster
    do {
        c++;    // check next cluster in FAT
        // check if reached last cluster in FAT, re-start from top
        if ( c >= fp->mda->maxcls)
            c = 0;

        // check if full circle done, media full
        if ( c == fp->ccls)
        {
            FError = FE_MEDIA_FULL;
            return FAIL;
        }
```

```
        // look at its value
        i = readFAT( fp, c);
        } while ( i!=0);     // scanning for an empty cluster


        // mark the cluster as taken, and last in chain
        writeFAT( fp, c, FAT_EOF);

        // if not first cluster, link current cluster to the new one
        if ( fp->ccls >0)
            writeFAT( fp, fp->ccls, c);

        // update the MFILE structure
        fp->ccls = c;

        return TRUE;
    } // allocate new cluster
```

分配新簇（非簇链的第一个簇）的时候，函数 newFAT()用来保证簇链各簇之间的连接，并且标记每个簇以达到合理的利用。为了达到这种功能，该函数使用了两个辅助函数 readFAT()和 writeFAT()。

```
unsigned readFAT( MFILE *fp, unsigned ccls)
// fp       MFILE structure
// ccls     current cluster
// return   next cluster value,
//          0xffff if failed or last
{
    unsigned p, c;
    LBA l;

    // get address of current cluster in fat
    p = ccls;
    // cluster = 0xabcd
    // packed as:     0    |    1    |    2    |  3    |
    // word p       0    1 |  2    3 | 4    5 |  6    7 |..
    //              cd   ab|  cd   ab| cd   ab|  cd   ab|

    // load the fat sector containing the cluster
    l = fp->mda->fat + (p >> 8 );    // 256 clusters per sector
    if ( !readSECTOR( l, fp->buffer))
        return 0xffff;  // failed

    // get the next cluster value
    c = ReadOddW( fp->buffer, ((p & 0xFF)<<1));

    return c;

} // readFAT
```

函数 writeFAT()用于更新 FAT 的内容，并保存所有的当前副本。

```
unsigned writeFAT( MFILE *fp, unsigned cls, unsigned v)
// fp       MFILE structure
// cls      current cluster
// v        next value
// return   TRUE if successful, or FAIL
{
    unsigned p;
    LBA l;

    // get address of current cluster in fat
    p = cls * 2; // always even
    // cluster = 0xabcd
    // packed as:     0    |    1    |    2    |  3    |
    // word p       0    1 |  2    3 | 4    5 | 6    7 |..
    //              cd   ab|  cd   ab| cd   ab| cd   ab|

    // load the fat sector containing the cluster
    l = fp->mda->fat + (p >> 9 );
    p &= 0x1fe;
    if ( !readSECTOR( l, fp->buffer))
        return FAIL;

    // get the next cluster value
    fp->buffer[ p] = v;        // lsb
    fp->buffer[ p+1] = (v>>8);// msb

    // update all FAT copies
    for ( i=0; i<fp->mda->fatcopy; i++, l += fp->mda->fatsize)
       if ( !writeSECTOR( l, fp->buffer))
           return FAIL;

    return TRUE;

} // writeFAT
```

最后，函数 fwriteM()和函数 fcloseM()使用函数 writeDATA()向存储设备中的扇区写入数据，并根据当前簇的序号计算出扇区的地址。

```
unsigned writeDATA( MFILE *fp)
{
    LBA l;

    // calculate lba of cluster/sector
    l = fp->mda->data + (LBA)(fp->ccls-2) * fp->mda->sxc + fp->sec;

    return ( writeSECTOR( l, fp->buffer));

} // writeDATA
```

14.2.13 测试整个文件 I/O 模块

现在来测试刚刚完成的整个模块的功能。就像前一个测试一样，使用前几章开发的串行通信模块“conu2.c”和提供 delayms()函数的相同的延时模块。这次，在文件系统设置好以后，打开一个源文件（可以是任何文件），然后将文件内容复制到一个在已标记的位置创建的新“目标”文件中。下面是 “writetest.c”的主文件代码。

```
/*
** WriteTest.c
**
*/

#include <p24fj128ga010.h>

#include "SDMMC.h"
#include "fileio.h"
#include "../delay/delay.h"

#include "../8 comm/conu2.h"

#define B_SIZE   1024

char data[B_SIZE];


int main( void)
{
    MFILE *fs, *fd;
    unsigned r;

    //initializations
    initU2();                   // 115,200 baud 8,n,1

    putsU2( "init");
    while( !detectSD());        // assumes SDCD pin is by default an input
    Delayms( 100);              // wait for card to power up

    if ( mount())
    {
        putsU2("mount");
        if ( (fs = fopenM( "source.txt", "r")))
        {
            putsU2("source file opened for reading");
            if ( (fd = fopenM( "dest3.txt", "w")))
            {
                putsU2("destination file opened for writing");
                do{
                    r = freadM( data, B_SIZE, fs);

                    r = fwriteM( data, r, fd);

                    putU2('.');

                } while( r==B_SIZE);

                fcloseM( fd);
                putsU2("destination file closed");
            }
            else
                putsU2("could not open the destination file");
```

```
            fcloseM( fs);
            putsU2("source file closed");
        }
        else
            putsU2("could not open the source file");

        unmount();
        putsU2("unmount");

    }
    else
        putsU2("mount failed");

    // main loop
    while( 1);
} // main
```

一定要确保用实验中实际复制到存储卡中的文件的文件名取代了源文件的文件名。

创建一个新项目（这次取名为"WriteTest"）之后，向项目窗口中添加所有必需的模块，包括：

- "sdmmc.c"；
- "fileio.c"；
- "conu2.c"；
- "delay.c"；
- "writetest.c"。

以及所有对应的 include 文件（.h）。

再次提醒读者，要遵循新项目和 ICD2 调试器的列表，但是这次要给堆增加一些空间，使得可以至少动态地为两个缓冲区和两个 MFILE 结构分配地址空间。

> **注解** 一旦为栈和全局变量预留了足够的空间之后，就没有理由再为堆留出任何存储空间了。为一个堆分配尽可能大的空间可以使得 malloc()和 free()最优地使用所有可用的空间。

构建项目，并对 Explorer16 板编程，现在可以运行测试了。如果顺利的话，不到一秒（实际时间取决于所选源文件的大小），读者就可以在终端机的屏幕上看到下面的一段信息：

```
init
mount
source file opened for reading
destination file opened for writing
.....................................................................................
destination file closed
source file closed
unmount
```

小圆点的数目与文件大小成比例，由于在本例的演示中选用的缓冲区大小为 1 024，因此每个小圆点将相应地被 1KB 的数据传输。这时，如果将 SD/MMC 的内容传送回读者的 PC 机上，

那么读者必须确保已经创建了一个新的文件，如图 14-8 所示。

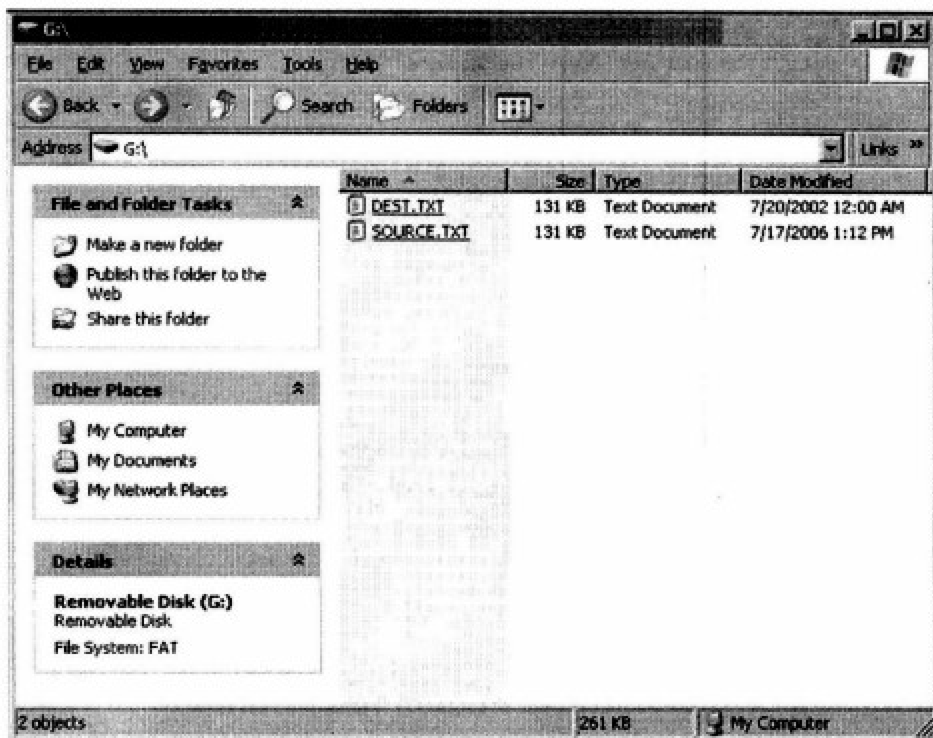


图 14-8 Windows Explorer 屏幕抓图

它的大小和内容与源文件相同，而日期和时间则为函数 fcloseM() 中设定的值。

注意，如果试图再一次运行这个测试程序的话，一定会失败的。

```
init
mount
source file opened for reading
could not open the destination file
source file closed
unmount
```

这是因为，在开发函数 fopenM() 的过程中，试图打开文件用来写入的时候，若在存储设备中找到一个已存在的相同名字的文件（DEST.TXT），则会出现错误报告。

注意，读者可以重新编译该程序，然后运行测试，数据缓冲区的大小可以从一个字节到 PIC24 所允许的存储空间的大小。函数 freadM() 和 fwriteM() 用来负责读者所要求的数据扇区的读和写操作。但完成操作所需时间将会有所改变。

14.2.14 代码大小

“WriteTest” 项目所生成的代码大小比第 13 章测试的 “sdmmc.c” 模块代码要大的多，如图 14-9 所示。

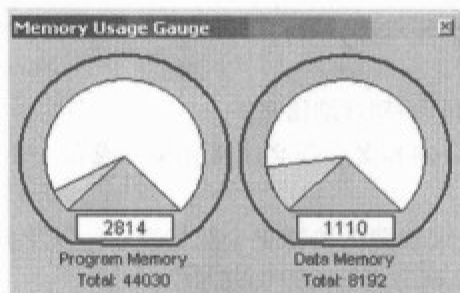


图 14-9 存储空间使用示意图

仍然关闭所有的最优化操作，代码大小仅仅是 8 442 个字节（2 814 个字×3）。这表明在 PIC24FJ128GA010 上只有 6%的程序存储空间可用。作者认为这只是实现强大功能所付出的一点点代价而已。

14.3 飞后小结

本章向读者介绍了基本的 FAT16 文件系统，并且开发了一个小型的接口模块，其允许 PIC24 向一个普通的大容量存储设备读取数据文件，或者向一个大容量存储设备写入数据文件。通过使用第 13 章为低级接口开发的“sdmmc.c”模块，创建一个 SD/MMC 存储卡的基本文件 I/O 接口。

现在读者可以在 PIC24 和任何一个支持 SD/MMC 卡的计算机系统（从 PDA 到手提式电脑和台式电脑，从 DOS、Windows、Linux 设备到使用 OS-X 的苹果电脑）之间共享数据了。

14.4 提示与技巧

嵌入式控制工程师经常问的一个问题是：“怎样才能切换到‘Thumb drive（U 盘）’（有时候是指一个 USB 棒），或者一个 USB 大容量存储设备，在程序和 PC 机之间共享/传输数据？”

最简短的答案是：“如果可以帮助它的话，就不要这样做！”

稍长一点的答案是：“使用 SD 卡！”以下就是原因。在第 13 章和本章中，相信读者已经发现使用 SD 卡读和写是非常简单的，而且只需要很少的代码和一个 SPI 端口（也可能是共用的）。

另一方面，从用户角度来说，USB 接口具有简单的外观和吸引力，但是对于一个中等的嵌入式控制设备，向 USB 接口的 U 盘读写数据确实非常复杂和费事。首先，SPI（高速同步串行口）接口的简单性将被 USB 总线接口的复杂性所取代。所需要的，不仅仅是使用标准的 USB 外围设备，还要使用 USB 主机的全部协议。硬件代价相对较高，比如专用的 USB 无线收发机、大型的串行接口引擎（SIE）。一个更大的开销是代码及其所需 RAM 存储空间。这些比之前所检测的 SD/MMC 卡的基本方案要复杂很多倍。

14.5 练习

(1) 考虑添加以下功能。

- ❑ 子目录管理。

- 擦除文件。
- 长文件名支持。

(2) 使用 RTCC 提供当前时间和日期信息。

(3) 考虑将当前 FAT 记录内容保存在高速缓存中（或者一个独立的缓冲区）以提高读写性能。

(4) 考虑一些修改，加入大数据块和/或整个簇的缓冲，以及多块读/写操作，用来优化 SD 卡的低级性能。其优点和缺点两方面都必须加以考虑。

14.6 推荐书目

- Buck,B. (2002)
  North Star Over My Shoulder
  Simon & Shuster, New York, NY
  一个描述飞行员一生经历的飞行故事。

14.7 网上链接

- http://www.tldp.org/LDP/tlk/tlk-title.html
  这是 David A. Rusling 的《Linux 内核》（*The Linux Kernel*）在线书籍，描述了 Linux 及其文件系统的内部工作情况。
- http://en.wikipedia.org/wiki/File_Allocation_Table
  这是维基百科中非常好的一页，介绍 FAT 技术的历史及其衍生技术。
- http://en.wikipedia.org/wiki/List_of_file_systems
  试图将所有目前使用的主要计算机文件系统进行明细列举和分类。

第 15 章 翱　翔

本章内容

- ▶在 PWM 模式下使用 PIC24 OC 模块
- ▶测试用作数模转换器的 PWM
- ▶产生模拟波形
- ▶话音信息再生
- ▶媒体播放器
- ▶WAVE 文件格式
- ▶函数 play()
- ▶低级音频例程
- ▶测试 WAVE 文件播放器
- ▶优化文件 I/O
- ▶LED 剖析
- ▶发掘更多

最后一次飞行是在 FAA（联邦航空局）主考官的监督下进行的飞行考试，当然会让人非常紧张外加一点害怕。这是前面所有飞行训练阶段的总结，学员必须将训练中所学的一切知识运用到实践中。不用担心——如果准备充分，它将是非常简单的，而且在学员还没有来得及察觉的时候它就很快地结束了。

就像最后的飞行考试一样，本章将会用到前面开发的多个模块，集成一个既实用又有娱乐性的演示项目：媒体播放器。

恭喜你成为了真正的飞行员，应该庆祝一番了！

15.1 飞行计划

本章将会再次使用 PIC24 输出比较模块来产生语音信号。当将脉冲宽度调制（PWM）和一些复杂的低通滤波器结合起来使用的时候，输出比较模块可以作为一个有效的数模转换器用来产生模拟输出信号。对模拟信号进行频率调制，使其频率处于人类可听范围之内，也就是大约 20 Hz 到 20 kHz 之间，那么声音就能被人听到了！

15.2 飞行

脉冲宽度调制的工作原理非常简单。脉冲是由一个定时器及其周期寄存器（period register）按照一定的时间间隔（T）产生。脉冲宽度（T_{on}）不是定值，而是可编程的，可以是定时器周期的 0 到 100%不等。脉冲宽度（T_{on}）和信号周期（T）之间的比称为占空比，如图 15-1 所示。

占空比的两个极端例子是：0%和 100%。前一种情况相当于信号总是关闭的，后一种情况则是输出信号总是开着的。介于这两种情况之间的脉宽对应的数（典型的是使用以 2 为底的对数表示的有限整数）通常被称为 PWM 的分辨率。比如，若一个信号含有 256 个脉冲宽度，则称这个 PWM 信号的分辨率为 8 位。

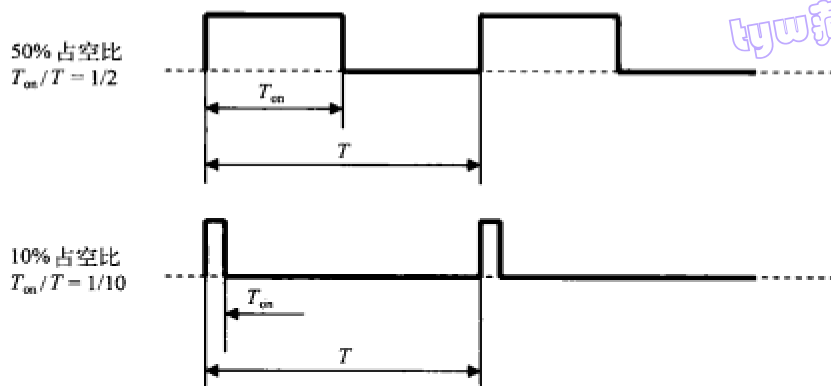


图 15-1　不同占空比的脉冲宽度调制（PWM）信号示例

如果向频谱分析仪输入一个理想的 PWM 信号并分析该信号的成分，将会发现它包含以下 3 个部分，如图 15-2 所示。

- 直流成分，幅值与占空比成比例。
- 基波频率的正弦成分（$f=1/T$）。
- 无限多的谐波，频率是基波频率的整数倍（$2f$，$3f$，$4f$，$5f$，$6f$…）。

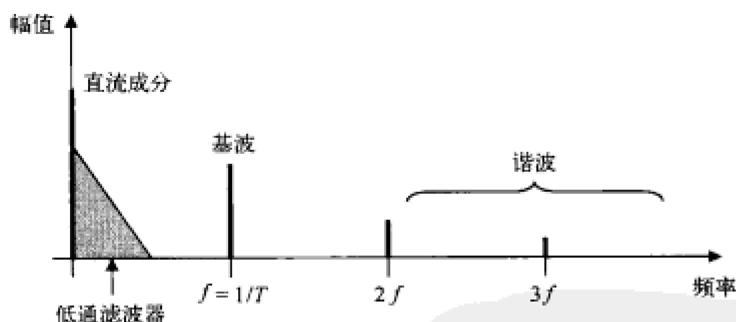


图 15-2　一个 PWM 信号的频谱分析

因此，如果能够将一个理想的低通滤波器连接到一个 PWM 信号发生器的输出端，以消除基波频率以外的频率分量，则可以得到一个直流模拟信号，其幅值直接与占空比成正比，如图 15-3 所示。

当然，这样的理想滤波器是不存在的，但是可以使用一些或简单或复杂的近似方法，以尽可能多地消除不需要的频率分量。这种滤波器可能与无源 R/C 电路（一阶低通滤波器）一样简单，也可能需要许多个（N 个）活动期（$2\times N$ 阶低通滤波）。

如果要产生音频信号，那么在选择 PWM 频率的时候，则可以利用人耳对频率的自然选择，这就像一个附加的滤波器，忽略了在 20 Hz 到 20 kHz 范围之外的所有频率。除此之外，在大多输出信号前加入的音频放大器，也需要在其输入端进行简单的滤波。换而言之，如果一个 PWM 信号工作在 20 kHz 或者 20 kHz 以上的频率时，这两种情况都会起一定作用，允许使用更简单更廉价的滤波器电路。

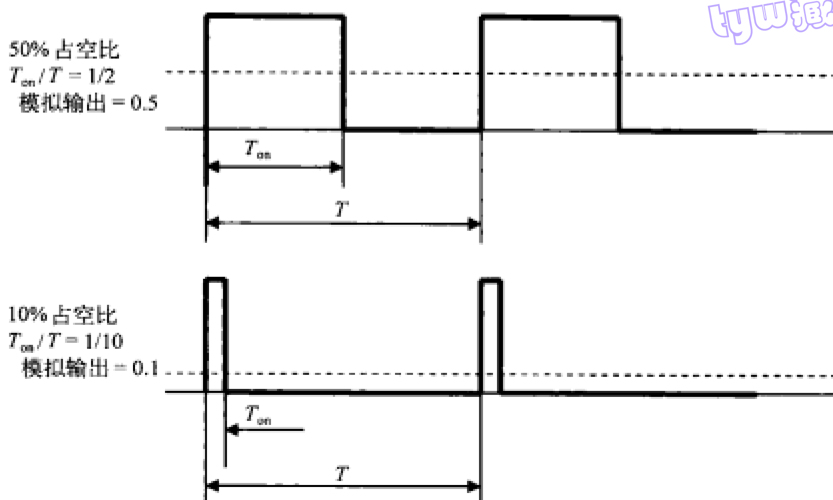


图 15-3 PWM 模拟输出和理想低通滤波电路

很明显，由于在每个 PWM 周期（T）内只能改变一次占空比，因此 PWM 的频率越高，输出模拟信号改变得就越快，继而产生的音频信号的频率就越高。

实际上，这意味着一个 PWM 可以产生的音频信号的最高频率只有 PWM 频率的一半。因此，一个 20 kHz 的 PWM 电路只能再生最高为 10 kHz 的音频信号，但是为了覆盖整个可听的音频频谱，需要频率至少为 40 kHz 的基本周期。这并不是巧合，比如说音乐 CD 是以 44 100 次每秒的采样速率进行数字化编码的。

15.2.1 在 PWM 模式下使用 PIC OC 模块

在前面的某章中已经使用过输出比较模块来产生精确的时间间隔（产生视频输出）。这次将在 PWM 模式下使用 OC 模块来产生连续的具有期望占空比的脉冲流，如图 15-4 所示。

高字节：

U-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
—	—	ICSIDL	—	—	—	—	—
位 15							位 8

低字节：

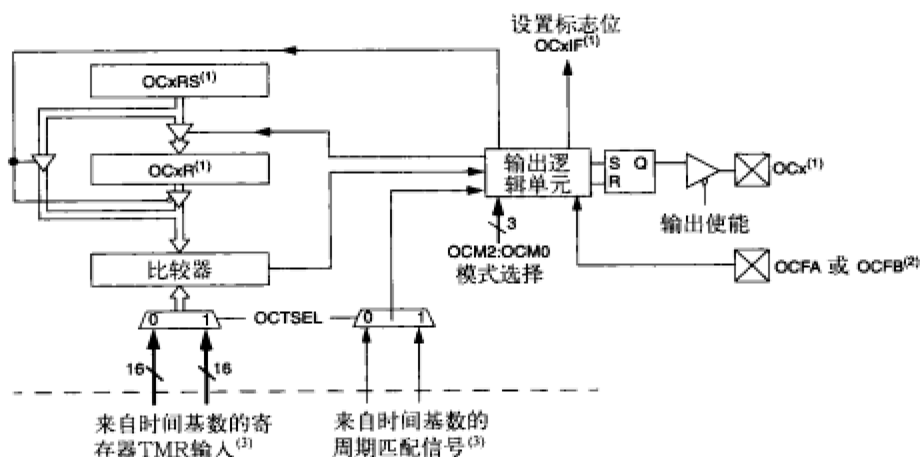
R/W-0	R/W-0	R/W-0	R-0, HC	R-0, HC	R/W-0	R/W-0	R/W-0
ICTMR	ICI1	ICI0	ICOV	ICBNE	ICM2	ICM1	ICM0
位 7							位 0

图 15-4 输出比较模块主控寄存器 OCxCON

要产生 PWM 信号，OC 模块的初始化就是将 OCxCON 控制寄存器中的 3 个 OCM 位设置成 0x110 配置。还有一种可用的 PWM 模式（0x111），但是对于发生故障的输入引脚没有作用，而只是作为一种保护机制（电机控制/功率转换）服务于一些应用中。接下来选择一个用于 PWM 周期基准的定时器。这只能在定时器 2 和定时器 3 之间选择，到目前为止两个定时器的作用没有区别。但对于不同的定时器设置，将会产生不同的结果（如图 15-5 所示）。

要记住至少需要产生 40 kHz 的 PWM 信号，并且假设使用 Explorer16 板上的一个 16 MHz

的外围时钟，这样就能计算出预分频器值和周期寄存器 PRx 的最佳值。将预分频器值设置为 1∶1 可以得到一段 400 周期的时间，从而产生一个精确的 40 kHz 的信号。这个值也反映了输出比较模块的占空比的分辨率。由于存在 400 个占空比值，并且在 256 (2^8) 和 512 (2^9) 之间，因此可以使用的分辨率为 8 位或者 9 位。将频率降低到 20 kHz 可能增加多一位分辨率（在 9 到 10 之间），但同时也会将输出频率限制在最大 10 kHz 的范围之内，对人耳产生的影响虽小但很显著。在设置好所选定的定时器之后，在写 OCxCON 寄存器之前，必须首次向寄存器 OCxR 和寄存器 OCxRS 中写入第一个占空比的值。在 PWM 模式中，这两个寄存器工作在主/从方式下。一旦 PWM 启动（在寄存器 OCxCON 中写入模式位），就只能通过改写寄存器 OCxRS 来改变占空比了。为了避免出现小故障，并留出整个周期（T）用来准备下一个占空比的值，寄存器 OCxR 只在每个周期的开始处从寄存器 OCxRS 复制新值，进行更新。



(1) 其中“x”表示使用 1~8 个输出比较通道对应的寄存器。
(2) OCFA 引脚控制 OC1~OC4 通道。OCFB 引脚控制 OC5~OC8 通道。
(3) 每个输出比较通道可使用两个可选的时间基准中的一个。关于模块的时间基数，请参阅设备数据表。

图 15-5 输出比较模块图

下面是 OC1 模块的一个简单的初始化例子：

```
void initAudio( void)
{
    // init TMR3 to provide the timebase
    T3CON = 0x8000;          // enable TMR3, prescale 1:1, internal clock
    PR3 = 400-1;             // set the period for the given bitrate
    _T3IF = 0;               // clear interrupt flag
    _T3IE = 1;               // enable TMR3 interrupt

    // init PWM
    // set the initial duty cycles (master and slave)
    OC1R = OC1RS = 200;          // init at 50%

    // activate the PWM module
    OC1CON = 0x000E;

} // initAudio
```

注意，借此机会开放了 Timer3 中断，以提醒一个新的周期开始，并且决定怎样更新 OC1RS 的下一个占空比值。

15.2.2 将 PWM 用作数/模转换器测试

为了能够在 Explorer16 板上进行测试，需要在原型区域添加一些独立的元件。使用一个 1 000 Ω的电阻和一个 100 nf 的电容可以组成一个最简单的低通滤波器（一阶低通滤波器，截止频率为 1.5 kHz）。将电阻和电容串联起来连接到 PORTD 的引脚 0 上设置的 OC1 模块的输出引脚上，如图 15-6 所示。

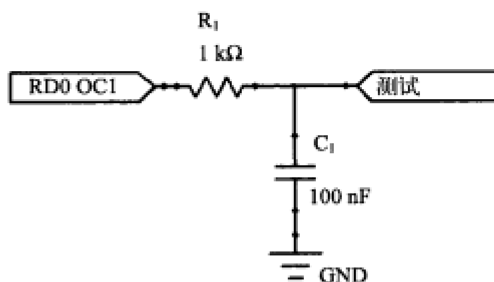


图 15-6 使用 PWM 信号产生模拟输出

只需要使用几行代码就可以完成这个小小的测试：

```
void _ISRFAST _T3Interrupt( void)
{
    // clear interrupt flag and exit
    _T3IF = 0;
} // T3 Interrupt


main( void)
{
    initAudio();

    // main loop
    while( 1);

}// main
```

添加常用的头文件和包含文件，将程序代码保存在一个新的文件——TestDA.c 中，就可以新建一个包含该文件的快速测试项目，并使用 ICD2 调试器来对 Explorer16 板编程。

将电表或者示波器探头（如果有条件的话）连接到测试端，运行程序并验证平均（直流）输出电压。

电表的指针或者示波器上的示数在大约 1.5V 处左右摆动，也就是 Explorer16 板上数字 I/O 引脚上的正常输出电压的 50%。这与初始化程序设置的占空比的值 200（400 周期的时间）是一致的。如果有示波器的话，可以用示波器的探头直接接触电阻 R_1 的另一端（直接接触 OC1 模块的输出端），可在屏幕上观察到频率为 40 kHz 且占空比为 50%的方波。

现在读者可以修改初始化程序，使用 0 到 399 之间不等的值来验证电路的响应和输出信号

随着占空比（0~3V 之间的模拟值）变化的比例。

15.2.3　产生模拟波形

在 OC1 模块的帮助下，读者已经跨越了由 0 和 1 组成的数字世界和在 0~3V 之间生成许多值的模拟世界的界限。

现在可以用占空比做游戏了。改变占空比，产生任意类型和形状的波形。首先将这个项目稍微改动一下，在中断子程序中原本空置的地方添加如下代码：

```
OC1RS = (count < 20) 400 : 0;
count++;
if ( count >= 40)
    count = 0;
```

读者必须将 count 声明为全局整型变量并初始化为 0。

保存并重新构建项目，在 Explorer16 板上测试代码。

每隔 20 个 PWM 周期，滤波器输出都会有从 3V（100%）到 0V（0%）的交替波形，在示波器的屏幕上产生一个频率为 1 kHz（40 kHz/40）的可视方波。

使用以下的算法可以产生一个非常有趣的波形：

```
OC1RS = count*10;
count++;
if ( count >= 40)
    count = 0;
```

这将产生一个峰值近似为 3V 的三角波（锯齿波），在第 40 步中占空比逐步从 0 变成 100%（每步 2.5%），然后突然回落到 0，重复下去。重复的频率也是 1 kHz。

虽然这两个例子产生的都是可识别的（基本的）高音调声音（大约 1 kHz），但是如果将它们输入音频放大器中，则没有一个例子可以发出“动听的”声音。在声音频谱中存在大量的可听到的谐波音频，这将使得声音中夹杂着不悦耳的嗡嗡声。

为了产生一个清晰的声音，读者需要一个纯粹的正弦曲线。以下的中断服务子程序能达到这个目标，将产生一个频率为 400 kHz 的完美正弦曲线（从音乐级别上来说，将会接近 A 级）。

```
void _ISRFAST _T3Interrupt( void)
{// compute the new sample for the next cycle
    OC1RS = 200+ ( 200* sin(count *0.0628));
    count++;
    if ( count >= 40)
        count = 0;
    // clear interrupt flag and exit
    _T3IF = 0;
} // T3 Interrupt
```

遗憾的是，按照 PIC24 和 C30 数学库的最快速度，读者没有机会使用 sin() 函数，执行期望的乘法和加法运算，并以 400 Hz 的期望频率获得新的占空比值。Timer3 每隔 25 μs 中断一次，对于一个复杂的浮点型运算来说，这个时间太短了，所以中断服务子程序将会“跳过”中断并产生一个频率只有期望值一半的（降低了一个音节）正弦型输出信号。但是，如果将这个信号输入音频放大器，读者将欣喜地发现声音的清晰度得到了显著改善。

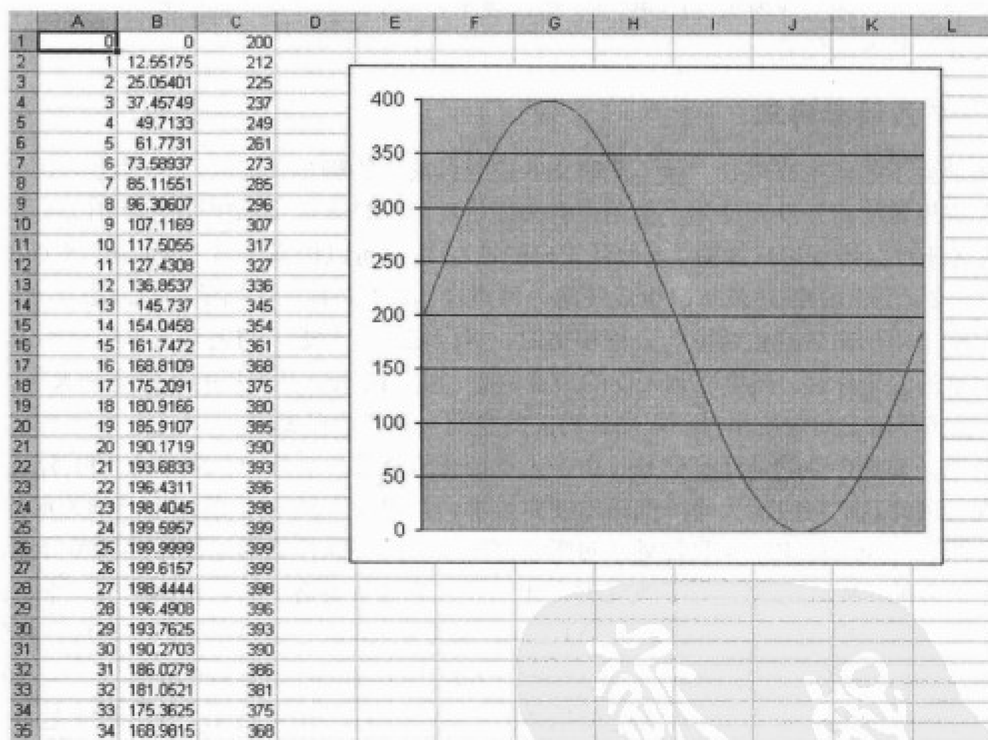
对于更高的频率，为了使得运行时的计算量最少（最好是工作在整数值上），需要将正弦值预先准备好。下面的例子使用了包含预先计算值的表格（存放在 PIC24 的 Flash 程序存储空间中）的例子。通过电子表格制作程序并使用下面的公式可以得到这个表格：

```
= offset + INT( amplitude * SIN( ROW * 6.28/ PERIOD))
```

对于一个含有 100 个样本（400 Hz），地址偏移量和幅值为 200 的周期，可以得到：

```
=200 + INT( 200*SIN(A1 *6.28/100))
```

如图 15-7 所示，向电子表格的第一列（A）中装入一个计数器值，复制公式并填入第二列（B）的前 100 行，将输出格式化为 0 的十进制数。



	A	B	C
1	0	0	200
2	1	12.56175	212
3	2	25.05401	225
4	3	37.45749	237
5	4	49.7133	249
6	5	61.7731	261
7	6	73.58937	273
8	7	85.11551	285
9	8	96.30607	296
10	9	107.1169	307
11	10	117.5065	317
12	11	127.4308	327
13	12	136.8537	336
14	13	145.737	345
15	14	154.0458	354
16	15	161.7472	361
17	16	168.8109	368
18	17	175.2091	375
19	18	180.9166	380
20	19	185.9107	385
21	20	190.1719	390
22	21	193.6833	393
23	22	196.4311	396
24	23	198.4045	398
25	24	199.5957	399
26	25	199.9999	399
27	26	199.6157	399
28	27	198.4444	398
29	28	196.4908	396
30	29	193.7625	393
31	30	190.2703	390
32	31	186.0279	386
33	32	181.0621	381
34	33	175.3625	375
35	34	168.9815	368

图 15-7 使用电子表格制作 400 Hz 正弦曲线的表格

然后，将整个列剪切粘贴到源代码中，依照 C 语言的规则在每行结尾处加上逗号。

```
void _ISRFAST _T3Interrupt( void)
{
    // load the new samples for the next cycle
    OC1RS = Table[ count];

    count++;
    if ( count >= 40)
        count = 0;

    // clear interrupt flag and exit
    _T3IF = 0;
} // T3 Interrupt
```

```
const int Table[100] = {
200,
212,
225,
237,
249,
...
149,
161,
174,
186,
199
};
```

这次能够容易地生成所要求的 400 Hz 输出频率，同时在定时器 3 的中断请求之间仍有足够的时间来处理其他的任务。

15.2.4 话音信息再生

只要学会了如何生成声音，就不能阻止我们继续前进的脚步。在嵌入式的世界中还有无穷的潜能和应用等待着开发和使用。使用声音来提供反馈、警告提示和错误提示可以吸引用户的注意，或者还可以增加用户体验（若做法正确的话），这样可以提高任何"人性化"界面的性能。事实上，只要有波形的描述就可以生成任何一种声音，而不必拘泥于简单的音调或基本的旋律。同上一个例子中用来装载正弦值的表格相类似，可以使用一个更大的表格来容纳一件乐器甚至是一个完整的音乐作品，产生准确无误的声音信息。唯一的限制是可用的存储空间大小，PIC24 在 Flash 程序存储空间中使用毗邻程序代码区的空间来保存数据表格。

特别地，如果存储的是话音信息，由于人类话音的能量主要集中在频率为 400 Hz 至 4 kHz 的范围内，因此我们可以大大地降低输出频率要求，并将 PWM 播放的速率限制为每秒 8 000 次采样。注意，为了使 PWM 谐波处于可听音频范围之外并保持低通滤波器的物美价廉，必须维持一个较高的 PWM 频率。只要改变占空比，并且降低从表格中读取新数据的速率，此处为每五个中断读取一次（即 40 000/8 000 = 5)。每秒采样 8 000 次，理论上可以将存储在控制器的 Flash 存储器中的声音信息播放长达 16 s。这对于单芯片方案已经是很好的结果了。为了提高或者加倍话音播放的潜在能力，可以考虑使用类似于 ADPCM 的压缩技术。ADPCM 是自适应差分编码调制的英文开头字母的缩写，它基于相邻两个采样样本之间的差异小于每个样本的绝对值的假设，因此可以使用更少的位进行编码。实际使用的位数得到了最优化，在保证期望的压缩率的前提下，动态地改变编码位数可以避免语音信号失真——因此，可以说成"自适应"。

15.2.5 媒体播放器

本章的剩余部分将会使用前面几章开发的所有库和各项功能，并集成在一个更大的工程中。这里尝试创造一个基本的多媒体播放器来播放 SD/MMC 存储卡上的立体声音乐文件。使用的格式为与大多数音频设备兼容的无压缩 WAVE 格式，同时也是从音乐CD中提取文件的默认目标格式。

首先使用常用的列表新建一个项目。接着，立即向项目源文件列表中添加 SD/MMC 低级接口和用于访问 FAT16 文件系的文件 I/O 库。

在打开要读取的文件之后，读者必须能够理解用来编码它所包含的数据的指定格式。

15.2.6 WAVE 文件格式

以 WAVE 格式编码的扩展名为“.WAV”的文件，是最简单的一种格式，但是仍然需要小心地使用。WAVE 格式是 RIFF 文件格式的一种变体。RIFF 文件格式是很多操作系统都可使用的一种标准文件格式，它采用一种特殊的技术将多片信息/数据分割成许多的“块”进行存储。块是一个头部含有两个 32 位元素（即块 ID 和块大小）的数据块，如表 15-1 所示。

表 15-1 数据块（块）的格式

地址偏移量	大 小	值	描 述
0x00	4	ASCII	块 ID
0x04	4	大小	块大小（内容大小）
0x08	大小		数据内容
0x08+size	1	0x00	可选择填充

注意，块的大小是 2 的倍数，这样 RIFF 文件中的所有数据都能够按字对齐。如果数据块大小不是 2 的整数倍，则需要在块的结束处添加一个额外的填充字节。

使用“RIFF” ID 的块通常出现在“.WAV”文件的开始处，其数据块以一个 4 字节的“类型”域开始。该类型域包含字符串“WAVE”。块可以像俄罗斯套娃一样进行嵌套，但是在一个给定类型的块中仍可能嵌套多个子块。

表 15-2 说明了一个“.WAV”文件的 RIFF 块结构。

表 15-2 “WAVE”类型的“RIFF”块

地址偏移量	大 小	值	描 述
0x00	4	“RIFF”	这是 RIFF 块 ID
0x04	4	大小	数据块的大小+4
0x08	4	“WAVE”	类型
0x10	大小		数据块（子块）

数据块包含一个“fmt”块，其后是一个“数据”块。通常，一幅图相当于一千个字。基本的 WAVE 文件格式如图 15-8 所示。

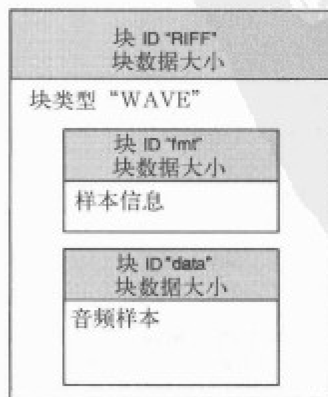


图 15-8 基本的 WAVE 文件格式

“fmt”块包含一组已定义好的参数用于充分描述跟在“数据”块之后的样本流，如表 15-3 所示。

表 15-3　“fmt”块的内容

偏 移 量	大　小	描　述	值
0x00	4	块 ID	“fmt”
0x04	4	块数据大小	16＋ 额外的格式字节
0x08	2	压缩编码	无符号整数
0x0a	2	通道数	无符号整数
0x0c	4	采样率	无符号长整型
0x10	4	平均字节/每秒	无符号长整型
0x14	2	块对齐	无符号整数
0x16	2	每次采样的有效位	无符号整数（>1）
0x18	2	额外的格式字节	无符号整数

在“fmt”和“data”块之间，还可以存在一些包含附加文件信息的其他块。因此用户必须查看块 ID 并浏览列表，直到找到所需要的块。

15.2.7　函数 play()

现在创建一个新的软件模块，用于打开一个给定的“.WAV”文件。在捕捉并译码“fmt”块中的信息之后，初始化音频输出模块。该模块与 15.2.1 节开发的音频输出模块相似。这里命名该软件模块为“wave.c”。

```
/*------------------------------------------------------------------------
** Wave.C
**
**  Wave File Player
**  Uses 2 x 8 bit PWM channels
**
*/
#include <stdlib.h>
#include "../Audio/Audio PWM.h"
#include "../sdmmc/sdmmc.h"
#include "../sdmmc/fileio.h"

// chunk ID definitions
#define  RIFF_DWORD  0x46464952UL
#define  WAVE_DWORD  0x45564157UL
#define  DATA_DWORD  0x61746164UL
#define  FMT_DWORD   0x20746d66UL


typedef struct {
    // data chunk
    unsigned long dlength;  // actual data size
    char     data[4];       // "data"

    // format chunk
    unsigned    bitpsample; //
```

```
    unsigned      bpsample;     // bytes per sample (4 = 16 bit stereo)
    unsigned long bps;          // bytes per second
    unsigned long srate;        // sample rate in Hz
    unsigned      channels;     // number of channels (1= mono, 2= stereo)
    unsigned      subtype;      // always 01
    unsigned long flength;      // size of encapsulated block (16)
    char      fmt_[4];          // "fmt_"

    char      type[4];          // file type name "WAVE"
    unsigned long tlength;      // size of encapsulated block
    char      riff[4];          // envelope "RIFF"
} WAVE;
```

WAVE 结构体对于在同一个位置收集所有的“fmt”参数是很有用的，块 ID 宏指令有助于用户识别不同的 ID，将它们看作长整型数，并且允许用户进行快速高效的比较。

接下来开始编写函数 play()。这里只需要一个参数——文件名。

```
unsigned play( const char *name)
{
    int       i;
    WAVE      wav;
    MFILE     *f;
    unsigned wi;
    unsigned long lc, r, d;
    int       skip, size, stereo, fix, pos;

    // 1. open the file
    if ( (f = fopenM( name, "r")) == NULL)
    {   // failed to open
        return FALSE;
    }
```

在打开文件并报告可能的错误之后，立即开始在数据缓冲区中查找 RIFF 块 ID 和 WAVE 类型 ID。作为一种标识，将用来确定是否打开了正确的文件：

```
// 2. verify it is a RIFF formatted file
if ( ReadL( f->buffer, 0) != RIFF_DWORD)
{
    fclose( f);
    return FALSE;
}

// 3. look for the WAVE type
if ( (d = ReadL( f->buffer, 8)) != WAVE_DWORD)
{
    fclose( f);
    return FALSE;
}
```

顺利的话，用户可以验证“fmt”块是否位于数据块内部的第一行。如果是的话，则用户将得到用于处理重播数据块的所有信息。

```
// 4. look for the chunk containing the wave format data
if ( ReadL( f->buffer, 12) != FMT_DWORD)
```

```
    return FALSE;

wav.channels    = ReadW( f->buffer, 22);
wav.bitpsample  = ReadW( f->buffer, 34);
wav.srate       = ReadL( f->buffer, 24);
wav.bps         = ReadL( f->buffer, 28);
wav.bpsample    = ReadW( f->buffer, 32);
```

接下来开始寻找“数据”块，检测“fmt”块结尾处的下一个数据块的块 ID 域，如果不匹配，则跳过整个块。

```
// 5. search for the data chunk
wi = 20 + ReadW( f->buffer, 16);
while ( wi < 512)
{
    if (ReadL( f->buffer, wi) == DATA_DWORD)
        break;
    wi += 8 + ReadW( f->buffer, wi+4);
}
if ( wi >= 512) // could not find a data chunk in current sector
{
    fclose( f);
    return FALSE;
}
```

如果在这个过程中访问了当前缓冲区中的所有数据，则意味出现了问题。典型的 WAV 文件是从音乐 CD 中提取数据得到的，紧跟在“fmt”块之后的只有“数据”块。其他的设备（比如 MIDI 接口）可能产生结构更加复杂的“WAV”文件，包含“数据”块、“播放列表”、“提示”、“标题”等。但是现在只关注播放一般类型的“WAV”文件。

一旦找到，“数据”块的大小将会反映文件中含有的采样样本的真实数目。

```
// 5.1 find the data size (actual wave content)
wav.dlength = ReadL( f->buffer, wi+4);
```

“播放”的快慢由播放的采样速率所决定。采样速率很有可能超出播放的能力，因此需要跳过一些采样点来降低数据速率。设定 48 K 次采样/秒的速率为上限，能够足够快速地读取数据以维持至少 8 位的分辨率。

```
// 6. compute the period and adjust the bit rate
r = wav.bps / wav.bpsample;     // r = samples per second
skip = wav.bpsample;            // skip factor to reduce bandwith (stereo)
while ( r > 48000)
{
    r >>= 1;                    // as you divide sample rate by two
    skip <<= 1;                 // multiply skip by two
}
```

对于更高速率，可以将采样速率除以 2 或者将跳跃速率加倍。

然后计算所要求的 PWM 的周期（用来设定寄存器 PRx）。如果该周期超过了寄存器的可用位数（16 位），则会出现问题，导致出现一个大于 65 536 的周期值。

```
// 6.1 check if the sample rate is compatible with the TMR3 prescaler 1:1
d = (16000000L/r)-1;
if ( d > ( 65536L))            // max TMR3 period value (16 bit)
{
    fclose( f);
    return FALSE;
}
```

在播放过程中，要记录从文件中提取出的样本数，用来确定何时到达文件结尾处。长整型变量 lc 用来为样本数做记录。

```
// 7. start loading the buffers
// determine the number of bytes composing the wav data chunk
lc = wav.dlength;
```

注意，到目前为止还没有使用函数 freadM()，因为读者已经窥见缓冲区，知道函数 freadM()早已载入其中了。

为了使播放更加流畅，可以使用双缓冲方案，即在音频中断服务子程序从一个缓冲区中读取数据的同时，向另一个缓冲区中填入新的数据。如图 15-9 所示，ABuffer[]实际上是被定义为两个 B_SIZE 字节大小的块。每个 B_SIZE 为 512 的整数倍，因此每次调用函数 freadM()都会覆盖整个数据扇区，从而达到最大的效率。必须保证使用函数 freadM()向一个缓冲区装填数据的时间少于 PWM 中断服务子程序播放另外一个缓冲区中所有数据所需的时间。

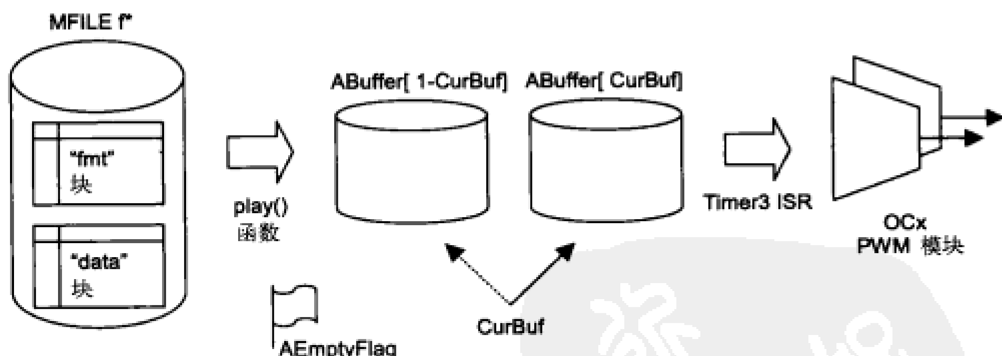


图 15-9 双缓冲方案示意图

当使用双缓冲方案时，装填两个缓冲区作为开始：

```
// 8. pre-load both buffers
r = fread( ABuffer[0], B_SIZE*2, f);
AEmptyFlag = FALSE;
lc-= B_SIZE*2 ;                 // we assume here that lc>=B_SIZE*2!!!
```

这里，假设.WAV 文件包含的数据至少可以填满两个缓冲区，但是如果用户打算使用少于几千字节数据的短文件的话，则需要修改代码。查看函数 freadM()返回的字节数，并在缓冲区结尾处补上正确的填充数。

现在做好准备可以初始化音频播放"机器"了，即调整函数 T3Interrupt()使之适用于立体声播放的两个通道。同时，增强跳跃采样样本的能力，使得在必要的时候能够降低采样速

率，并且增加处理 16 位样本（有符号）和 8 位样本（无符号）的能力。所有这些信息都将以参数列表的形式传递给音频模块 initAudio()：

```
// 9. start playing, enable the interrupt
initAudio( wav.srate, skip, size, stereo, fix, pos);
```

一旦定时器中断被激活，服务程序就立即开始消费第一个缓冲区中的数据，并且在所有的内容都被消费完毕后，标志位 AEmptyFlag 置 1，以提示用户需要从 WAV 文件中提取新的数据，同时另外一个缓冲区将被激活。因此，为了保证播放流畅，可以使用一个小循环，不断检查 AEmptyFlag 是否准备好以再次装填数据，记录从文件中读取的字节数直到所有字节使用完毕。

```
// 10. keep feeding the buffers in the playing loop
while (lc >=B_SIZE)
{
    if ( AEmptyFlag)
    {
        r = fread( ABuffer[1-CurBuf], B_SIZE, f);
        AEmptyFlag = FALSE;
        lc-= B_SIZE;

    }
} // while wav data available
```

事实上，在文件中剩下的数据不足以装满整个缓冲区的时候，程序早就停止运行了。此时，除非数据块大小是缓冲区大小的整数倍并且没有其他的新数据可读取，否则，最后的数据需要经过扩充，才能装填入最后一个播放的缓冲区中。

```
// 11. flush the buffers with the data tail
if( lc>0)
{
    // load the last sector
    r = fread( ABuffer[1-CurBuf], lc, f);
    last = ABuffer[1-CurBuf][r-1];
    while(( r<B_SIZE) && (last>0))
        ABuffer[1-CurBuf][r++] = last;

    // wait for current buffer to be emptied
    AEmptyFlag = 0;
    while (!AEmptyFlag);
}
```

等待最后一个缓冲区播放完毕，然后立即终止音频播放：

```
// 12. finish the last buffer
AEmptyFlag = 0;
while (!AEmptyFlag);

// 13.stop playback
haltAudio();
```

关闭文件，释放存储空间，返回调用程序：

```
    // 14. close the file
    fclose( f);

    // 15. return with success
    return TRUE;

} // play
```

为了完成这个模块，必须创建一个小的包含文件“wave.h”，用来声明函数 play()的原型：

```
/*------------------------------------------------------------------------
** Wave.H
**
**  Wave File Player
**  Uses 2 x 8 bit PWM channels
**
*/

unsigned play( const char *name);
```

15.2.8 低级音频程序

刚刚完成的函数 play()非常依赖低级音频模块，以完成实际的定时器和 OC 外围设备的初始化，以及执行实际的 PWM 占空比的周期性更新。将该模块命名为“audiopwm.c”，它主要是以本章开始时开发的代码为基础，并逐步扩展到管理立体声播放的两个声道和一些附加选项。同时使用 OC1 和 OC2 模块，产生左声道和右声道。定时器中断服务子程序是音频播放功能的真正核心。由于在每个周期中，需要用光数据，并向 PWM 模块输入新的样本，因此指针 BPtr 可以用来追踪每个缓冲区中的位置。

```
void _ISRFAST _T3Interrupt( void)
{
    // 1. load the new samples for the next cycle
    OC1RS = 30+(*BPtr ^ Fix);
    if ( Stereo)
        OC2RS =30 + (*(BPtr + Size) ^ Fix);
    else    // mono
        OC2RS = OC1RS;
```

指针的推进由大量字节来提供，这取决于样本的大小（16 位或者 8 位）和跳跃采样的需要。当必须使用 play()函数时，可以实现降低采样速率的目的：

```
// 2. skip samples to reduce the bitrate
BPtr += Skip;
```

一旦载入缓冲区的数据使用完毕，则交换当前活跃的缓冲区：

```
// 3. check if buffer emptied
if ( --BCount == 0)
{
    // 3.1 swap buffers
    CurBuf = 1- CurBuf;
    BPtr = ABuffer[ CurBuf];
```

重新设定样本计数器，并设置一个标志以提示 play()程序，在再次使用完数据之前需要

准备好新的缓冲区。

```
        // 3.2 restart counter
        BCount = B_SIZE/Size;

        // 3.3 flag a new buffer needs to be filled
        AEmptyFlag = 1;
    }
```

只有在中断标志位清零后才退出：

```
        // 4. clear interrupt flag and exit
        _T3IF = 0;

    } // T3 Interrupt
```

回想本章开始创建的初始化程序，读者会发现，除了有更多的参数需要传递给所调用的应用程序和复制给模块本身（私有）变量之外，这里的初始化程序还是同样地简单易懂：

```
void initAudio( long bitrate, int skip, int size, int stereo, int fix, int pos)
{
    // 1. init pointers
    CurBuf = 0;                    // start with buffer0 active first
    BPtr = ABuffer[ CurBuf]+pos;
    BCount = (B_SIZE-pos)/size;  // number of samples to be played
    AEmptyFlag = 0;
    Skip = skip;
    Fix = fix;
    Stereo = stereo;
    Size = size;
```

选择一个缓冲区作为正在使用的“当前”缓冲区，初始化所有的指针和计数器。然后初始化定时器，其中断机制为：

```
    // 2. init the timebase
    T3CON = 0x8000;          // enable TMR3, prescale 1:1, internal clock
    PR3 = FCY / bitrate;     // set the period for the given bitrate
    Offset = PR3/2;
    _T3IF = 0;               // clear interrupt flag
    _T3IE = 1;               // enable TMR3 interrupt
```

将占空比初始化为周期值的一半，用来提供平均 50%的初始输出电平。

```
    // 3. set the initial duty cycles
    OC1R = OC1RS = Offset;   // left
    OC2R = OC2RS = Offset;   // right
```

最后，输出比较模块被激活：

```
    // 4. activate the PWM modules
    OC1CON = 0x000E;         // CH1 and CH2 in PWM mode, TMR3 based
    OC2CON = 0x000E;

} // initAudio
```

在音频播放部分最后调用的函数 haltAudio()无疑是最简单的。它唯一的任务就是关闭

Timer3，从而冻结输出比较模块。使用它们的整个中断机制是：

```
void haltAudio( void)
{
    T3IE = 0;               // disable TMR3 interrupt
} // halt audio
```

完成这个模块还需要添加常用的文件头部，包含文件和全局变量的定义，这自然包括了音频缓冲区。

```
/*
** Audio PWM demo
**
*/

#include <p24fj128ga010.h>

#include "AudioPWM.h"

#define _FAR __attribute__(( far))

// global definitions
unsigned Offset;                        // 50% duty cycle value
char  _FAR  ABuffer[ 2][ B_SIZE];       // double data buffer
int      CurBuf;                        // index of buffer in use
volatile int AEmptyFlag;                // flag a buffer needs to be filled

// internal variables
int Stereo;                             // flag stereo play back
int Fix;                                // sign fix for 16-bit samples
int Skip;                               // skip factor to reduce sample/rate
int Size;                               // sample size (8 or 16-bit)

// local definitions
unsigned char *BPtr;                    // pointer inside active buffer
int BCount;
```

注意，和第 14 章的做法一样，当为视频应用程序分配大的缓冲区时，可以使用 `far` 属性以分配 PIC 近地址空间以外的存储空间。

为了能够使用 PWM 模块提供的服务，include 文件 “`audiopwm.h`” 将声明 “`wave.c`” 模块和其他应用程序需要的所有定义和原型。

```
/*
** AudioPWM.h
**
*/

#define FCY      16000000L              // instruction cycle frequency
#define TCYxUS   16                     // number of Tcycles in a microsecond
#define B_SIZE   2048                   // audio buffer size


extern char ABuffer[ 2][ B_SIZE];       // double data buffer
```

```
extern int CurBuf;                    // index of buffer in use
extern volatile int AEmptyFlag;       // flag a buffer needs to be filled

void initAudio( long bitrate, int skip, int size, int stereo, int fix, int pos);
void haltAudio( void);
```

15.2.9　测试 WAVE 文件播放器

低级音频模块和播放模块都已经完成，现在可以将它们整合起来，通过播放一些音乐来测试。

新建一个项目“WaveTest”并添加一些必要的模块和 include 文件：

- “sdmmc.c”；
- “fileio.c”；
- “audiopwm.c”；
- “wave.c”；
- “sdmmc.h”；
- “fileio.h”；
- “audiopwm.h”；
- “wave.h”。

现在，添加一个主模块“wavetest.c”，它只包含几行代码。它将调用函数 play()，指明将要复制到 SD/MMC 卡上的唯一文件的文件名（TRACK00.WAV）。

```
/*
**   WaveTest
**
*/

#include <p24fj128ga010.h>

#include "SDMMC.h"
#include "fileio.h"

#include "../Audio/Audio PWM.h"
#include "../Wave/Wave.h"


main( void)
{

    TRISA = 0xff00;

    if ( !mount())
        PORTA = FError + 0x80;
    else
    {
        if ( play( "TRACK00.WAV"))
           PORTA = 0;
      else
           PORTA = 0xFF;
```

```
    } // mounted

    while( 1)
    {
    } // main loop

} //main
```

发光二极管 LED 的 PORTA 排是用来显示错误报告的：要么是 mount()操作失败，要么是在存储设备上未找到文件。

使用适当的列表在 Explorer16 板上构建项目和执行代码。不要忘记为堆（heap）预留一部分空间，因为模块 fileio.c 是使用堆（heap）来为缓冲区和数据结构分配空间的。

为了循序渐进地进行，作者建议用户使用逐渐增大采样速率和样本大小的 WAV 文件来测试程序。例如，第一次测试，使用 8 位样本、单声道、采样速率为 8 K 样本每秒的 WAV 文件。继续进行，则逐渐增加格式的复杂度和播放的速度，一直到最后一次测试，即测试这个程序的所有功能时，使用 16 位样本、立体声、采样速率为 44 100 样本每秒的文件。之所以采用逐渐增加的方式，是为了确定模块“fileio.c”的性能是否能够达到最终的要求。实际上，随着采样速率、通道个数和样本大小的增加，文件系统所需的带宽也会增加。可以很快地计算出这些参数的不同组合所要求的不同性能水平。

表 15-4 中列出了各种文件格式所需的字节速率——每秒钟播放函数消耗的字节数（样本大小×通道数×采样速率）。特别地，最后一列显示的是一个存满数据的新缓冲区需要间隔多长时间再次被填充（字节速率为 512），即 play()程序从 WAV 文件中读取下一扇区数据的可用时间。

表 15-4 各种文件格式所需的字节速率

文件	样本大小	通道	采样速率	字节速率	重载周期（ms）
单声道	1	1	8 000	8 000	64.0
立体声	1	2	8 000	16 000	32.0
8 位单声道	1	1	22 050	22 050	23.2
8 位立体声	1	2	22 050	44 100	11.6
8 位高位率单声道	1	1	44 100	44 100	11.6
8 位高位率立体声	1	2	44 100	88 200	5.8
16 位单声道	2	1	44 100	88 200	5.8
16 位立体声	2	2	44 100	176 400	2.9

注意，由于工作在 44 100 Hz 频率时，PIC24 PWM 产生的分辨率低于 9，所以音频 PWM 模块只能设计成使用 16 位样本的最高有效位。因此，播放上表中最后两种格式的任意一种时，音频输出的质量根本不会提高，所得到的只是对 SD/MMC 卡存储空间的一种浪费。如果用户想要最大化地利用可用存储空间，那么应该确保在复制文件到存储卡上的时候，将样本大小降低为 8 位。这样，对于相同的输出音频分辨率，将能够在卡上存储双倍数量的文件。

建议读者顺着上表从上往下地试验一遍，将会发现在超过一定的范围（很可能为超过音频 8 位、高位率、单声道）之后，音频就不能正常播放了。播放的时候会跳过、重复、打嗝等，

总之不是正常的声音。这是因为函数 freadM()已经达到了极限，不能够满足播放的需求了。装载一个新缓冲区的平均时间超过了消耗一个缓冲区的时间，因此一段时间之后，play()程序开始跟不上进度，音频播放函数开始重复缓冲区的内容或者播放还没有填充完毕的缓冲区。

15.2.10　优化文件 I/O

在编写文件 I/O 库函数的时候，甚至更早以前编写访问 SD/MMC 卡的低级函数的时候，关注的重点始终只是完成任务，而从来没有真正评估过程序的性能水平。也许现在有必要研究研究它了。为了使得每一个例子都能够在免费的 MPLAB C 编译器学生版本上可用，本书的其他部分避免使用编译器的任何优化性能，并且一直默认如此。也许使用一点技巧就可以进一步提高性能。

第一件事就是弄清楚 PIC24 从存储卡中读取数据时，在什么地方花的时间最多。检查函数 freadM()会发现只有两个低级子程序被调用。一个是函数 readDATA()，用来从当前簇载入一个新扇区；另一个是函数 nextFAT()，用来在当前簇中的扇区被消耗完毕的时候识别下一个簇。这两个函数轮流地调用函数 readSECTOR()，检索一个 512 字节的数据块。最后，调用标准 C 语言函数 memcpy()，向应用程序传递一个数据块。所以函数 freadM()的最终性能是依赖于 readSECTOR()和 memcpy()的性能的。

15.2.11　LED 剖析

如果使用示波器，能相对容易地判断出两者中哪一个的影响更大。事实上，也许读者记得，当初设计函数了 readSECTOR()，是为了使用 PORTA 上的一个 LED 来作为 SD/MMC 卡上读操作正在执行的指示信号。如果在播放循环中将示波器连接到 LED 的阳极，读者将能够看到一个周期性脉冲，脉冲宽度表明传输数据过程中 PIC24 花费在函数 readSECTOR()上的准确时间，如图 15-10 所示。两个脉冲之间的停顿时间，与 PIC24 在函数 memcpy()内和 freadM()函数调用栈剩余的大部分时间是成正比的。只要瞥一瞥，读者立即就会发现问题所在。

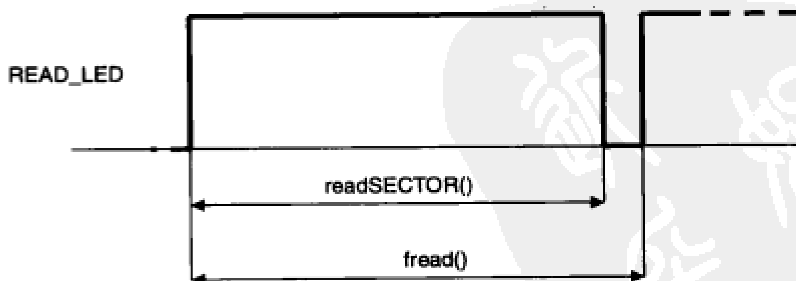


图 15-10　将示波器连接在 READ_LED 端口上

毫无疑问，函数 readSECTOR()需要读者认真关注。它占用了一个周期的大部分时间，超过 10 ms！

```
int readSECTOR( LBA a, char *p)
// a         LBA of sector requested
// p         pointer to sector buffer
// returns   TRUE if successful
```

```
{
    int r, tout;

    READ_LED = 1;

    r = sendSDCmd( READ_SINGLE, ( a << 9));
    if ( r == 0)    // check if command was accepted
    {
        // wait for a response
        tout = 10000;
        do{
            r = readSPI();
            if ( r == DATA_START) break;
        }while( --tout>0);

        // if it did not timeout, read a 512 byte sector of data
        if ( tout)
        {
            for( i=0; i<512; i++)
                *p++ = readSPI();

            // ignore CRC
            readSPI();
            readSPI();

        } // data arrived

    } // command accepted

    // remember to disable the card
    disableSD();
    READ_LED = 0;

    return ( r == DATA_START);      // return TRUE if successful
} // readSECTOR
```

在函数列表中，读者可以发现，PIC24 只在以下三个可能的地方花费很多的时间。

(1) 函数 sendSDCmd()。

(2) 等待从存储卡（可能是个慢速的 SD/MMC 卡）中取出 DATA_START 的循环。

(3) 从卡中逐字节地读取共计 512 字节的循环。

要区分以上三种情况，只需要通过移动其中的一个函数到 READ_LED 开启和关闭的括号内，就能识别。如果重新编译这个项目并运行几次测试的话，读者将会发现如果将函数 sendSDCmd() 放入括号中，脉冲时间将会缩小成基本不可读的一点。

```
READ_LED = 1;
r = sendSDCmd( READ_SINGLE, ( a << 9));
READ_LED = 0;
```

这表明存储卡能足够快地响应命令，因此时间一定是耗费在其他地方了。

如果将等待从卡中取出 DATA_START 的循环放入括号中，结果非常地相似：

```
READ_LED = 1;
    // wait for a response
    tout = 10000;
    do{
        r = readSPI();
        if ( r == DATA_START) break;
    }while( --tout>0);
READ_LED = 0;
```

这是第三个循环，虽然看起来无伤大雅但已重复了 512 次，似乎占用了 PIC24 剩余的所有周期。

```
READ_LED = 1;
        for( i=0; i<512; i++)
            *p++ = readSPI();
READ_LED = 0;
```

这里才是将所有优化措施集中的地方。

第一步是试着取消对函数 readSPI()的调用，取而代之的是，直接使用下面的几行内联代码：

```
READ_LED = 1;
   for( i=0; i<512; i++)
   {
       SPI2BUF = 0xFF;                          // write to buffer for TX
       while( !(SPI2STATbits.SPIRBF));          // wait for transfer complete
       *p++ = SPI2BUF;                          // read the received value
   }
READ_LED = 0;
```

如果能够耐心地重新建立项目，然后测量新的脉冲宽度，用户将会看到性能的一些改进，但是改进并不明显。

15.2.12　发掘更多

下一步自然是查看编译器是如何使用那几行代码的。快速浏览反汇编列表窗口中的一个特殊的代码段。

```
139:                              for( i=0; i<512; i++)
 011A4   EB0000     clr.w 0x0000
 011A6   980750     mov.w 0x0000,[0x001c+10]
 011A8   9000DE     mov.w [0x001c+10],0x0002
 011AA   201FF0     mov.w #0x1ff,0x0000
 011AC   508F80     sub.w 0x0002,0x0000,[0x001e]
 011AE   3C0013     bra gts, 0x0011d6
 011CE   90005E     mov.w [0x001c+10],0x0000
 011D0   E80000     inc.w 0x0000,0x0000
 011D2   980750     mov.w 0x0000,[0x001c+10]
 011D4   37FFE9     bra 0x0011a8
142:                              {
144:                                  SPI2BUF = 0xFF;
 011B0   200FF0     mov.w #0xff,0x0000
 011B2   881340     mov.w 0x0000,0x0268
146:                                  while( !SPI2STATbits.SPIRBF);
```

```
 011B4   BFC260      mov.b 0x0260,0x0000
 011B6   FB8000      ze.b 0x0000,0x0000
 011B8   600061      and.w 0x0000,#1,0x0000
 011BA   E00000      cp0.w 0x0000
 011BC   32FFFB      bra z, 0x0011b4
147:                                   *p++ = SPI2BUF;
 011BE   4701E4      add.w 0x001c,#4,0x0006
 011C0   780093      mov.w [0x0006],0x0002
 011C2   801340      mov.w 0x0268,0x0000
 011C4   784100      mov.b 0x0000,0x0004
 011C6   780001      mov.w 0x0002,0x0000
 011C8   784802      mov.b 0x0004,[0x0000]
 011CA   E80081      inc.w 0x0002,0x0002
 011CC   780981      mov.w 0x0002,[0x0006]
148:                               }
... <<for loop closing code here>>
 011D6
```

执行一个看起来简单明了的 for 循环却使用了超过 25 条指令。自然地，应当想办法减少最内层循环（while 循环）的复杂度，它用来等待 SPI 外围设备完成数据传输任务。尽管该循环的大部分都是简单直接的，但是内部仍然有一个标志扩展（ze.b）指令看起来像是冗余的。这让人怀疑它可能不仅仅只是编译器用来检测 SPI2STARTbit.SPIRBF 标志位所使用的位域算法的副产物。

直接屏蔽 SPI2STAT 寄存器的内容，重新格式化代码，改进条件并排除可疑。

```
for( i=0; i<512; i++)
{
    SPI2BUF = 0xFF;                  // write to buffer for TX
    while( !(SPI2STAT & 1));         // wait for transfer complete
    *p++ = SPI2BUF;                  // read the received value
}
```

虽然现在的代码只比以前的少了一条指令，但是要知道这条指令在 512 个循环的每个循环中要重复至少两次。

```
139:                             for( i=0; i<512; i++)
 011A4   EB0000      clr.w 0x0000
 011A6   980750      mov.w 0x0000,[0x001c+10]
 011A8   9000DE      mov.w [0x001c+10],0x0002
 011AA   201FF0      mov.w #0x1ff,0x0000
 011AC   508F80      sub.w 0x0002,0x0000,[0x001e]
 011AE   3C0012      bra gts, 0x0011d4
 011CC   90005E      mov.w [0x001c+10],0x0000
 011CE   E80000      inc.w 0x0000,0x0000
 011D0   980750      mov.w 0x0000,[0x001c+10]
 011D2   37FFEA      bra 0x0011a8
142:                             {
144:                                   SPI2BUF = 0xFF;
 011B0   200FF0      mov.w #0xff,0x0000
 011B2   881340      mov.w 0x0000,0x0268
145:                                   while( !(SPI2STAT&1));
 011B4   801300      mov.w 0x0260,0x0000
 011B6   600061      and.w 0x0000,#1,0x0000
```

```
 011B8   E00000      cp0.w 0x0000
 011BA   32FFFC      bra z, 0x0011b4
146:                              *p++ = SPI2BUF;
 011BC   4701E4      add.w 0x001c,#4,0x0006
 011BE   780093      mov.w [0x0006],0x0002
 011C0   801340      mov.w 0x0268,0x0000
 011C2   784100      mov.b 0x0000,0x0004
 011C4   780001      mov.w 0x0002,0x0000
 011C6   784802      mov.b 0x0004,[0x0000]
 011C8   E80081      inc.w 0x0002,0x0002
 011CA   780981      mov.w 0x0002,[0x0006]
147:                          }
... <<for loop closing code here>>
 011D4
```

下一个锦囊妙计是设置一个特殊的寄存器，用来保存经常使用的变量，从而尽量减少数据在软件栈中的进出频率。候选者之一是变量 i，它是 for 循环的下标，另外一个是指针 p。

C30 编译器要求使用以下的语法为寄存器声明一个变量：

```
register unsigned i asm( "w5");
```

但是，除非指定的寄存器是可用的，否则结果就是不确定的。通常编译器将前四个寄存器 w0…w3 用作便签式存储器，用户不能使用它们。同时，该寄存器不能作为函数的参数，同指针 p 一样地不幸，它有可能影响调用函数的寄存器分配方案。为了快速消除这种限制，可以将指针 p 的内容复制给一个新的指针 q，其代码如下：

```
register unsigned i asm( "w5");
register char * q asm("w6");
q = p;
for( i=0; i<512; i++)
{
    SPI2BUF = 0xFF;
    while( !(SPI2STAT&1));  // wait for transfer to complete
    *q++ = SPI2BUF;         // read the received value
}
```

这次，重新编译代码，读者可以观察到外层循环的大小已大为减少，for 循环编码也得到了精简：

```
139:                         for( i=0; i<512; i++)
 011A6   EB0280      clr.w 0x000a
 011A8   201FF0      mov.w #0x1ff,0x0000
 011AA   528F80      sub.w 0x000a,0x0000,[0x001e]
 011AC   3E000D      bra gtu, 0x0011c8
 011C4   E80285      inc.w 0x000a,0x000a
 011C6   37FFF0      bra 0x0011a8
142:                         {
144:                             SPI2BUF = 0xFF;
 011AE   200FF0      mov.w #0xff,0x0000
 011B0   881340      mov.w 0x0000,0x0268
145:                             while( !(SPI2STAT&1));
 011B2   801300      mov.w 0x0260,0x0000
 011B4   600061      and.w 0x0000,#1,0x0000
 011B6   E00000      cp0.w 0x0000
 011B8   32FFFC      bra z, 0x0011b2
```

```
146:                                    *q++ = SPI2BUF;
 011BA   801340      mov.w 0x0268,0x0000
 011BC   784080      mov.b 0x0000,0x0002
 011BE   780006      mov.w 0x000c,0x0000
 011C0   784801      mov.b 0x0002,[0x0000]
 011C2   E80306      inc.w 0x000c,0x000c
... <<for loop closing code here>>
 011C8
```

现在只剩下 17 条指令了。最后一步是试着使用另外一种不同类型的循环来计数 512 个字节的数据。使用一个简单的 do 循环从前向后计数：

```
register unsigned i asm( "w5");
register char * q asm("w6");
q = p;
i = 512;
do {
    SPI2BUF = 0xFF;
    while( !(SPI2STAT&1));  // wait for transfer to complete
    *q++ = SPI2BUF;         // read the received value
} while ( --i>0);
```

这是到现在为止最好的结果——只包含 15 条指令！

```
 011A6   202005      mov.w #0x200,0x000a
141:                              do{
144:                                  SPI2BUF = 0xFF;
 011A8   200FF0      mov.w #0xff,0x0000
 011AA   881340      mov.w 0x0000,0x0268
145:                                  while( !(SPI2STAT&1));
 011AC   801300      mov.w 0x0260,0x0000
 011AE   600061      and.w 0x0000,#1,0x0000
 011B0   E00000      cp0.w 0x0000
 011B2   32FFFC      bra z, 0x0011ac
146:                                  *q++ = SPI2BUF;
 011B4   801340      mov.w 0x0268,0x0000
 011B6   784080      mov.b 0x0000,0x0002
 011B8   780006      mov.w 0x000c,0x0000
 011BA   784801      mov.b 0x0002,[0x0000]
 011BC   E80306      inc.w 0x000c,0x000c
148:                              } while (--i>0);
 011BE   E90285      dec.w 0x000a,0x000a
 011C0   E00005      cp0.w 0x000a
 011C2   3AFFF2      bra nz, 0x0011a8
 011C4
```

是时候重新在 Explorer16 板上运行新的代码了。再次检测一遍函数 readSECTOR()完成读取一个 512 K 字节的数据扇区所需要的时间。读者将会惊喜地发现，现在可以将所需时间减少到 1.5 ms 之内了。这样的话，用户完全可以播放一个严重损坏的 WAV 文件了。

15.3 飞后小结

这最后一章应该是学习经历的最理想总结了，它将最先进的软件和硬件设施融合在一个项

目中，并且覆盖了数字领域和模拟领域。首先使用输出比较模块外围设备产生可听音频域的模拟信号。然后将这一技术与第 14 章中开发的“fileio.c”模块结合起来，用来播放大容量存储设备（SD/MMC 卡）上的无压缩音乐文件（WAV 文件格式）。本章得到的基本媒体播放器仅仅是一个新的起点，这项工程有着无限的拓展空间。如果想要满足好奇心和想象力的话，读者可以尽情、无约束地使用 PIC24 和编译器 MPLAB C30。

15.4 提示与技巧

播放过程的开头和结尾是 PWM 模块最重要的两个部分。静止的时候，输出滤波电容放电，输出电压为 0V。一旦播放开始，50%的工作周期将会迫使输出电压猛增到 1.5 V 左右，同时产生一个响亮的、不和谐的咔咔声。反之，在播放的结尾处应当关闭 PWM 模块，而不是像示范例子中一样只关掉中断。这种现象与模拟放大器在上电和关闭时发生的现象很相似。有一个解决的办法是添加几行代码。在定时器中断使能和播放机启动之前，加入一个小型（短时间）循环，用来逐渐地将输出占空比从 0 增加到从播放缓冲器中取出的第一个样本值的大小。

15.5 练习

(1) 研究用于声音信息的 ADPCM 信号的解码技术（详见应用说明 AN643）。
(2) 查找卡上所有的 WAV 文件并建立一个播放列表。
(3) 使用伪随机数发生器实现“shuffle”模式并逐渐清空播放列表。
(4) 使用基本数字滤波技术去除不希望的频率，增强其他频率或者只是使声音和语音失真。

15.6 推荐书目

- Mandrioli, D. & Ghezzi, C. (1987)
  Theoretical Foundations of Computer Science
  John Wiley & Sons, New York, NY
  这本书并不易读，但是如果读者对计算机科学的更深层次的数学以及理论基础感兴趣的话，不妨一读。
- Leroy Cook, (1990)
  101 Things to Do With Your Private License
  TAB 图书，McGraw-Hill 公司的分支

15.7 网上链接

- http://en.wikipedia.org/wiki/RIFF
  RIFF 文件的格式解释。
- http://en.wikipedia.org/wiki/WAV
  WAV 文件的格式解释。
- http://ccrma.stanford.edu/courses/422/projects/WaveFormat/
  WAVE 文件格式的另一个出色的描述。

Programming 16-bit Microcontrollers in C
Learning to Fly the PIC 24

16位单片机C语言编程 基于PIC24

新型16位PIC24芯片为嵌入式工程师提供了比以往PIC微控制器速度更快、存储容量更大、数量更多的外围接口，在功能非常强大的重要PIC设计应用中极具潜力。本书向读者详细介绍了PIC24芯片的编程、测试、调试等有关知识。

作者是Microchip公司的一位PIC专家，他对PIC革命性的技术具有独到的洞察力。本书从16位体系结构基础，讲到最复杂的编程场景，一步一步地引导读者进行学习。书中还介绍了大量C语言编程的实例，展现了在使用新型PIC芯片时如何巧妙地避免常见故障，高效地解决现实设计问题并优化程序代码，有经验的PIC用户和相关领域的新手都能从中获益。

读者将会掌握下面的知识和技术：

- 基本时序和I/O操作；
- 所有新硬件外设；
- 控制LCD显示；
- 产生音频和视频信号；
- 访问大容量的存储介质；
- 与PC共享大容量存储介质上的文件；
- 在Explorer 16演示板上进行实验；
- MPLAB-SIM和ICD2工具的调试方法。

Lucio Di Jasio 嵌入式控制系统设计专家，在PIC架构设计方面具有丰富的经验。曾任职于Microchip公司，对其产品性能以及开发流程都非常熟悉。除了本书外，他还著有《32位单片机C语言编程：基于PIC32》一书。

李中华博士 现任中山大学信息科学与技术学院讲师、硕士生导师。主要研究兴趣为嵌入式系统及自动化、智能系统与先进控制。

张雨浓博士 现任中山大学信息科学与技术学院“百人计划”教授、博士生导师，2007年入选教育部新世纪优秀人才支持计划。主要研究领域为冗余机器人、递归神经网络、高斯过程、科学计算和软硬件开发。

黄晓红副教授 现任广东轻工职业技术学院副教授，2006年入选广东省高等学校“千百十工程”校级培养对象。主要研究领域为工业自动化、楼宇智能控制技术。

本书译自原版*Programming 16-bit Microcontrollers in C: Learning to Fly the PIC24*，并由Elsevier授权出版。









本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010) 51095186
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议：电子电气/单片机
人民邮电出版社网址 www.ptpress.com.cn

ISBN 978-7-115-22149-0
定价：49.00元